Theory of Computation- Lecture Notes

Michael Levet

January 10, 2024

Contents

1 Mathematical Preliminaries

1.1 Set Theory

Definition 1 (Set)**.** A *set* is collection of distinct elements, where the order in which the elements are listed does not matter. The size of a set S, denoted $|S|$, is known as its *cardinality* or *order*. The members of a set are referred to as its elements. We denote membership of x in S as $x \in S$. Similarly, if x is not in S, we denote $x \not\in S$.

Example 1. Common examples of sets include the set of real numbers $\mathbb{R}$, the set of rational numbers $\mathbb{Q}$, and the set of integers $\mathbb{Z}$. The sets $\mathbb{R}^+, \mathbb{Q}^+$ and $\mathbb{Z}^+$ denote the strictly positive elements of the reals, rationals, and integers respectively. We denote the set of natural numbers $\mathbb{N} = \{0, 1, \ldots\}$. Let $n \in \mathbb{Z}^+$ and denote $[n] = \{1, \ldots, n\}$.

We now review several basic set operations, as well as the power set. It is expected that students will be familiar with these constructs. Therefore, we proceed briskly, recalling definitions and basic examples intended solely as a refresher.

Definition 2. Set Union Let A, B be sets. Then the *union* of A and B, denoted $A \cup B$ is the set:

$$A \cup B := \{x : x \in A \text{ or } x \in B\}$$

Example 2. Let $A = \{1, 2, 3\}$ and $B = \{4, 5, 6\}$. Then $A \cup B = \{1, 2, 3, 4, 5, 6\}$.

Example 3. Let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$. So $A \cup B = \{1, 2, 3, 4, 5\}$. Recall that sets do not contain duplicate elements. So even though 3 appears in both A and B, 3 occurs exactly once in $A \cup B$.

Definition 3. Set Intersection Let A, B be sets. Then the *intersection* of A and B, denoted $A \cap B$ is the set:

$$A \cap B := \{x : x \in A \text{ and } x \in B\}$$

Example 4. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \cap B = \{1, 3\}$. Now let $C = \{4\}$. So $A \cap C = \emptyset$.

Definition 4 (Symmetric Difference)**.** Let A, B be sets. Then the *symmetric difference* of A and B, denoted $A \triangle B$ is the set:

$$A \triangle B := \{x : x \in A \text{ or } x \in B, \text{ but } x \not\in A \cap B\}$$

Example 5. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \triangle B = \{2, 5\}$.

For our next two definitions, we let U be our *universe*. That is, let U be a set. Any sets we consider are subsets of U.

Definition 5 (Set Complementation)**.** Let A be a set contained in our universe U. The *complement* of A, denoted A^C or $\overline{A}$, is the set:

$$\overline{A} := \{x \in U : x \not\in A\}$$

Example 6. Let $U = [5]$, and let $A = \{1, 2, 4\}$. Then $\overline{A} = \{3, 5\}$.

Definition 6 (Set Difference)**.** Let A, B be sets contained in our universe U. The *difference* of A and B, denoted $A \setminus B$ or $A - B$, is the set:

$$A \setminus B = \{x : x \in A \text{ and } x \not\in B\}$$

Example 7. Let $U = [5]$, $A = \{1, 2, 3\}$ and $B = \{1, 2\}$. Then $A \setminus B = \{3\}$.

Remark: The Set Difference operation is frequently known as the *relative complement*, as we are taking the complement of B relative to A rather than with respect to the universe U.

Definition 7 (Cartesian Product)**.** Let A, B be sets. The *Cartesian product* of A and B, denoted $A \times B$, is the set:

$$A \times B := \{(a, b) : a \in A, b \in B\}$$

Example 8. Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Then $A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$.

Definition 8 (Power Set)**.** Let S be a set. The *power set* of S, denoted 2^S or $\mathcal{P}(S)$, is the set of all subsets of S. Formally:

$$2^S := \{A : A \subseteq S\}$$

Example 9. Let $S = \{1, 2, 3\}$. So $2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Remark: For finite sets S, $|2^S| = 2^{|S|}$; hence, the choice of notation.

Definition 9 (Subset)**.** Let A, B be sets. A is said to be a *subset* of B if for every $x \in A$, we have $x \in B$ as well. This is denoted $A \subseteq B$. Note that B is a *superset* of A.

Example 10. Let $A = [3], B = [6], C = \{2, 3, 5\}$. So we have $A \subseteq B$ and $C \subseteq B$. However, $A \not\subseteq C$ as $1 \not\in C$; and $C \not\subseteq A$, as $5 \not\in A$.

Remark: Let S be a set. The subset relation forms a partial order on 2^S. To show two sets A and B are equal, we must show $A \subseteq B$ and $B \subseteq A$. We demonstrate how to prove two sets are equal below.

Proposition 1.1. *Let $A = \{6n : n \in \mathbb{Z}\}, B = \{2n : n \in \mathbb{Z}\}, C = \{3n : n \in \mathbb{Z}\}$. So $A = B \cap C$.*

Proof. We first show that $A \subseteq B \cap C$. Let $n \in \mathbb{Z}$. So $6n \in A$. We show $6n \in B \cap C$. As 2 is a factor of 6, $6n = 2 \cdot (3n) \in B$. Similarly, as 3 is a factor of 6, $6n = 3 \cdot (2n) \in C$. So $6n \in B \cap C$. We now show that $B \cap C \subseteq A$. Let $x \in B \cap C$. Let $n_1, n_2 \in \mathbb{Z}$ such that $x = 2n_1 = 3n_2$. As 2 is a factor of x and 3 is a factor of x, it follows that $2 \cdot 3 = 6$ is also a factor of x. Thus, $x = 6n_3$ for some $n_3 \in \mathbb{Z}$. So $x \in A$. Thus, $B \cap C \subseteq A$. Thus, $A = B \cap C$, as desired. □

Proposition 1.2. *Let A, B, C be sets. Then $A \times (B \cup C) = (A \times B) \cup (A \times C)$.*

Proof. Let $(x, y) \in A \times (B \cup C)$. If $y \in B$, then $(x, y) \in (A \times B)$. Otherwise, $y \in C$ and so $(x, y) \in (A \times C)$. Thus, $A \times (B \cup C) \subseteq (A \times B) \cup (A \times C)$. Now let $(d, f) \in (A \times B) \cup (A \times C)$. Clearly, $d \in A$. So f must be in either B or C. Thus, $(d, f) \in A \times (B \cup C)$, which implies $(A \times B) \cup (A \times C) \subseteq A \times (B \cup C)$. We conclude that $A \times (B \cup C) = (A \times B) \cup (A \times C)$. □

1.2 Relations and Functions

Definition 10 (Relation)**.** Let X be a set. A k-ary relation on X is a subset $R \subseteq X^k$.

Example 11. The notion of equality $=$ over $\mathbb{R}$ is the canonical example of a relation. It is perhaps the most well-known instance of an *equivalence relation*, which will be discussed later.

Intuitively, a k-ary relation R contains k-tuples of elements from X that share common properties. Computer scientists and mathematicians are interested in a number of different relations, including the adjacency relation (graph theory), equivalence relations, orders (such as partial orders), and functions. In this section, functions, asymptotics, and equivalence relations will be discussed.

1.2.1 Functions

The notion of a *function* will be introduced first. Functions are familiar mathematical objects, which appear early on in mathematics education with the notion of an input-output machine. Roughly speaking, a function takes an input and produces an output. Some common examples include the linear equation $f(x) = ax + b$ and the exponential $f(x) = 2^x$.

We denote a function as follows. Let X and Y be sets. A function is a map $f : X \to Y$ such that for every $x \in X$, there is a unique $y \in Y$ where $f(x) = y$. We say that X is the *domain* and Y is the *codomain*. The *range* or *image* is the set $f(X) = \{f(x) : x \in X\}$. More formally, a function is defined as follows:

Definition 11. Function Let X and Y be sets. A function f is a subset (or 1-place relation) of $X \times Y$ such that for every $x \in X$, there exists a unique $y \in Y$ where $(x, y) \in f$.

Let's consider some formal functions and one example of a relation that is not a function.

Example 12. Let $f : \mathbb{R} \to \mathbb{R}$ be given by $f(x) = x$. This is known as the *identity map*.

Example 13. Let $g : \mathbb{R} \to \mathbb{R}$ be given by $g(x) = 3x^2$.

Example 14. Let $h : \mathbb{R} \to \mathbb{R}$ given by:

$$h(x) = \begin{cases} x & : x \neq 3 \\ -3, 2 & : x = 3 \end{cases}$$

Note that h is *not* a function as $(3, -3) \in h$ and $(3, 2) \in h$. The definition of a function states that there must be a *unique* y such that $(3, y) \in h$. If we revise h such that $h(3) = -3$ *only*, then h satisfies the definition of a function.

From a combinatorial perspective, special types of functions known as *injections* and *surjections* are of great importance. The idea is that if we have two sets X and Y and know the cardinality of X, then an injection or surjection from X to Y yields results about Y's cardinality. We can deduce similar results about Y's cardinality.

An injection is also known as a *one-to-one* function. Recall the definition of a function states that the map $f : X \to Y$ maps each $x \in X$ to a unique $y \in Y$. It allows for functions such as $g : \mathbb{R} \to \mathbb{R}$ given by $g(x) = 0$. Clearly, every $x \in \mathbb{R}$ maps to the same y-coordinate: $y = 0$. An injection disallows functions such as these. The idea is that each $y \in Y$ can be paired with at most one $x \in X$, subject to the constraint that each element in X must be mapped to some element from Y. So there can be unmapped elements in Y, but not in X.

We define this formally as follows.

Definition 12 (Injection)**.** A function $f : X \to Y$ is said to be an injection if $f(x_1) = f(x_2) \implies x_1 = x_2$. Equivocally, f is an injection if $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$.

Let's consider examples of functions that are injections, as well as those that fail to be injections.

Example 15. Let X be a set. Recall the identity map $\text{id} : X \to X$ given by $\text{id}(x) = x$. This function is an injection. Let $\text{id}(x_1) = \text{id}(x_2)$. Then we have $\text{id}(x_1) = x_1 = \text{id}(x_2) = x_2$, which implies that $x_1 = x_2$.

Example 16. Consider the function $g : \mathbb{R} \to \mathbb{R}$ be given by $g(x) = x^2$. Observe that g fails to be an injection. Let $g(x_1) = g(x_2) = 4$. We have $x_1 = 2$ and $x_2 = -2$, both of which map to 4. If we instead consider $h : \mathbb{R}^+ \to \mathbb{R}$ by $h(x) = x^2$, we have an injection since we only consider the positive real numbers. Observe as well that both g and h do not map to any element less than 0.

Remark: Let's reflect on what we know about injections. An injection is a function, in which any mapped element in the codomain is mapped to exactly once. There may be elements in the codomain which remain unmapped. As a result, for two sets X and Y, it is defined that $|X| \leq |Y|$ if there exists an injection $f : X \to Y$. Intuitively speaking, an injection pairs each element from the domain with an element in the codomain, allowing for leftover elements in the codomain. Hence, X has no more elements than Y if there exists an injection $f : X \to Y$.

Surjections or onto functions center around the codomain, rather than the domain. Recall the earlier example of $g : \mathbb{R} \to \mathbb{R}$ given by $g(x) = x^2$, which satisfies $g(x) \geq 0$ for every $x \in \mathbb{R}$. So the negative real numbers will never be maped under g. Surjections exclude functions like g. Intuitively, a function is a surjection if every element in the codomain is mapped. Any element of the codomain can have multiple domain points mapping to it, as long as each has at least one domain point mapping to it. We define this formally as follows.

Definition 13 (Surjection)**.** Let X and Y be sets. A function $f : X \to Y$ is a surjection if for every $y \in Y$, there exists an $x \in X$ such that $f(x) = y$.

We have already seen an example of a function that is not a surjection. Let us now consider a couple examples of functions that are surjections.

Example 17. Recall the identity map $\text{id} : X \to X$. For any $x \in X$, we have $\text{id}(x) = x$. So the identity map is a surjection.

Example 18. Let $X = \{a, b, c, d\}$ and let $Y = \{1, 2, 3\}$. Define $f : X \to Y$ by $f(a) = f(b) = 1, f(c) = 2$ and $f(d) = 3$. This function is a surjection, as each $y \in Y$ is mapped under f. Observe that there are more X elements than Y elements. If X instead had two elements, then f would not be a surjection because at most two of the three elements in Y could be mapped.

Remark: Similarly, let's now reflect upon what we know about surjections. A surjection is a function in which every element of the codomain is mapped at least once. Some elements in the codomain may have multiple elements in the domain mapping to them. Therefore, if there exists a surjection $f : X \to Y$, then $|X| \geq |Y|$. We now introduce the notion of a bijection. A function is a bijection if it is both an injection and a surjection. Intuitively, a bijection matches the elements in the domain and codomain in a one-to-one manner. That is, each element in the domain has precisely one mate in the codomain and vice versa. For this reason, two sets X and Y are defined to have the same cardinality if there exists a bijection $f : X \to Y$. Combinatorialists use bijections to ascertain set cardinalities. The idea is that given sets X and Y, with $|Y|$ known, can we construct a bijection $f : X \to Y$? If the answer is yes, then $|X| = |Y|$.

Definition 14 (Bijection)**.** Let X and Y be sets. A bijection is a function $f : X \to Y$ that is both an injection and a surjection.

Example 19. Some examples of bijections include the identity map, as well as the linear equation $f(x) = mx + b$.

We conclude by showing that the composition of two injective functions are injective, and that the composition of two surjective functions are surjective. This implies that the composition of two bijections is itself a bijection, which is an important fact when working with permutations (which we shall see later).

Proposition 1.3. *Let $f : X \to Y$ and $g : Y \to Z$ be injective functions. Then $g \circ f$ is also injective.*

Proof. Let $x_1, x_2 \in X$ be distinct. As f is injective, $f(x_1) \neq f(x_2)$. Similarly, as g is injective, $g(f(x_1)) \neq g(f(x_2))$. So $(g \circ f)(x_1) \neq (g \circ f)(x_2)$, as desired. As x_1, x_2 were arbitrary, we conclude that $g \circ f$ is injective. □

Proposition 1.4. *Let $f : X \to Y$ and $g : Y \to Z$ be surjective functions. Then $g \circ f$ is also surjective.*

Proof. Let $z \in Z$. As g is surjective, there exists $y \in Y$ such that $g(y) = z$. Now as f is surjective, there exists $x \in X$ such that $f(x) = y$. Thus, $(g \circ f)(x) = z$. As z was arbitrary, it follows that $g \circ f$ is surjective. □

1.2.2 Equivalence Relations

Equivalence relations are of particular importance in mathematics and computer science. Intuitively, an equivalence relation compares which elements in a set X share some common property. The goal is to then partition X into equivalence classes such that all the elements in one of these parts are all equivalent to each other. This allows us to select an arbitrary distinct representative from each equivalence class and consider only that representative.

This idea of partitioning comes up quite frequently. The integers modulo n, denoted $\mathbb{Z}/n\mathbb{Z}$, is a canonical example. Big-Theta is another important equivalence relation. Equivalence relations allow us to prove powerful theorems such as Fermat's Little Theorem from Number Theory and Cauchy's Theorem from Group Theory, as well as to construct a procedure to minimize finite state automata via the Myhill-Nerode Theorem.

In order to guarantee such a partition, an equivalence relation must satisfy three properties: reflexivity, symmetry, and transitivity. We define these formally below, restricting attention to binary relations.

Definition 15 (Reflexive Relation)**.** A relation R on the set X is said to be reflexive if $(a, a) \in R$ for every $a \in X$.

Definition 16 (Symmetric Relation)**.** A relation R on the set X is said to be symmetric if $(a, b) \in R$ if and only if $(b, a) \in R$ for every $a, b \in X$.

Definition 17 (Transitive Relation)**.** A relation R on the set X is said to be transitive if for every $a, b, c \in X$ satisfying $(a, b), (b, c) \in R$, then $(a, c) \in R$.

Definition 18 (Equivalence Relation)**.** An equivalence relation is a reflexive, symmetric, and transitive relation.

Let us break each of these definitions down and compare them to how the equality relation on $\mathbb{R}$ behaves. Intuitively, a real number is equal to itself; i.e., $3 = 3$, $0 = 0$ and $1 \neq 2$. The properties of an equivalence relation reflect this behavior. The reflexive axiom states that $(a, a) \in R$ for every $a \in X$. Intuitively, reflexivity captures this notion that an element is equivalent to itself.

Now consider the definition of a symmetric relation: for every $a, b \in X$, $(a, b) \in R$ if and only $(b, a) \in R$. Suppose in a high school algebra problem we deduce that that for the variables x and y, we have $x = y$. Does it make sense that $y \neq x$? Of course not. Equivalence relations must capture this property as well, which is the purpose of the symmetry axiom. Elements in the same equivalence class must be pairwise equivalent.

The last axiom is transitivity. We refer back to the example of the high school algebra problem. Suppose this time we have three variables x, y and z satisfying $x = y$ and $y = z$. Over the real numbers, it makes perfect sense that $x = z$. So from an intuitive perspective, it is important that equivalence relations enforce this property. However, from a more technical perspective, transitivity implies that the equivalence classes are pairwise disjoint. In other words, transitivity is really the driving force in partitioning the set into equivalence classes. This will be proven later.

The congruence relation $a \equiv b \pmod{n}$ is a canonical example of an equivalence relation. We prove this below.

Definition 19 (Congruence Relations)**.** Let $n \geq 1$ be an integer. The congruence relation modulo n is a binary relation on $\mathbb{Z}$ given by: $a \equiv b \pmod{n}$ (read as: a is congruent to b modulo n) if and only if n divides $b - a$.

Proposition 1.5. *Let $n \geq 1$ be an integer. The relation $a \equiv b \pmod{n}$ is an equivalence relation.*

Proof. We show that the congruence relation modulo n is reflexive, symmetric, and transitive.

- **Reflexivity.** Let $a \in \mathbb{Z}$. We show that $a \equiv a \pmod{n}$. So n divides $a - a = 0$. Thus, $a \equiv a \pmod{n}$. So the congruence relation modulo n is reflexive.
- **Symmetry.** Let $a, b \in \mathbb{Z}$ such that $a \equiv b \pmod{n}$. We show that $b \equiv a \pmod{n}$. Let $q \in \mathbb{Z}$ such that $nq = a - b$. Thus, $n(-q) = b - a$, so n divides $b - a$. Thus, $b \equiv a \pmod{n}$. So the congruence relation modulo n is symmetric.
- **Transitivity.** Let $a, b, c \in \mathbb{Z}$ such that $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$. We show that $a \equiv c \pmod{n}$. By the definition of the congruence relation, n divides $(a - b)$ and n divides $(b - c)$. Let $h, k \in \mathbb{Z}$ such that $nh = a - b$ and $nk = b - c$. So $nh + nk = n(h + k) = a - c$. Thus, n divides $a - c$, so $a \equiv c \pmod{n}$. It follows that the congruence relation modulo n is transitive.

We conclude that the congruence relation modulo n is an equivalence relation. □

We now formalize the notion of an equivalence class, with the goal of showing that an equivalence relation *partitions* a set. Informally, a partition of a set X is a collection of disjoint subsets of X, whose union is precisely X. This is formalized as follows.

Definition 20 (Partition)**.** Let X be a set. A *partition* of X is a set $\mathcal{P}$ satisfying the following.

- Each member $P \in \mathcal{P}$ is a non-empty subset of X.
- For any two distinct $P_1, P_2 \in \mathcal{P}$, $P_1 \cap P_2 = \emptyset$.
- $\displaystyle\bigcup_{P \in \mathcal{P}} P = X$.

Definition 21 (Equivalence Class)**.** Let X be a set, and let $\equiv$ be an equivalence relation on X. Fix $x \in X$. The *equivalence class* of x is the set $[x] = \{s \in X : x \equiv s\}$. That is, $[x]$ is the set of elements that are equivalent to x.

Remark: Note that $[x] = [s]$ for any $s \in [x]$. This follows from the transitivity of $\equiv$, which will be proven shortly.

Example 20. Fix $n \geq 1$. The equivalence classes of the congruence relation modulo n are the classes $[0], [1], \dots, [n-1]$. Some additional tools from number theory are required to justify this, so we omit a proof. Informally, the congruence relation modulo n is represented by the remainder classes upon division by n.

As an equivalence relation is reflexive, every element of the set belongs to some equivalence class. Thus, in order for an equivalence relation to partition the set, it suffices to show that the equivalence classes are pairwise disjoint.

Proposition 1.6. *Let $\equiv$ be an equivalence relation on the set X. Let $[x], [y]$ be distinct equivalence classes under $\equiv$. Then $[x] \cap [y] = \emptyset$.*

Proof. Suppose to the contrary that $[x] \cap [y] \neq \emptyset$. As $[x]$ and $[y]$ are distinct and have non-empty intersection, there exists $z \in [y] - [x]$. Without loss of generality, suppose $y \in [x] \cap [y]$. So $x \equiv y$. Since $z \in [y]$, we have $y \equiv z$. By transitivity, $x \equiv z$, which implies $z \in [x]$, a contradiction. □

1.3 Proof by Induction

Many theorems of interest ask us to prove a proposition holds for all natural numbers. Verifying such statements for all natural numbers is challenging, due to the fact that our domain is not just large but infinite. Informally, proof by induction allows us to verify a small subset of base cases. Together, these base cases imply the subsequent cases. Thus, the desired theorem is proven as a result.

Intuitively, we view the statements as a sequence of dominos. Proving the necessary base cases knocks (i.e., proves true) the subsequent dominos (statements). It is inescapable that all the statements are knocked down; thus, the theorem is proven true.

The most basic form of induction is the Principle of Weak Induction.

Definition 22 (Principle of Weak Induction)**.** Let $P(n)$ be a proposition regarding an integer n, and let $k \in \mathbb{Z}$ be fixed. If:

(a) $P(k)$ holds; and

(b) for every $m \geq k$, $P(m)$ implies $P(m+1)$,

then for every $n \geq k$, $P(n)$ holds.

The definition of the Principle of Weak Induction in fact provides a format for structuring proofs.

- First, we verify a single base case: for some $k \in \mathbb{N}$, the statement $P(k)$ holds.
- Second, we assume that $P(m)$ holds for some integer $m \geq k$. This step is known as the **inductive hypothesis**, and it is indispensible for a proof by induction. We must and do use the fact that $P(m)$ is true when proving that $P(m+1)$ holds.
- In our final step, we show that for an arbitrary $m \geq k$ that $P(m)$ implies $P(m+1)$. This is known as the **inductive step**.

We illustrate this proof technique with the following example.

Proposition 1.7. *Fix $n \in \mathbb{N}$. We have:* $\displaystyle\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Proof. We prove this theorem by induction on $n \in \mathbb{N}$.

- **Base Case.** Our first step is to verify the base case: $n = 0$. In this case, we have $\sum_{i=0}^{n} i = 0$. Note as well that $\frac{0 \cdot 1}{2} = 0$. Thus, the proposition holds when $n = 0$.

- **Inductive Hypothesis.** Now for our inductive hypothesis: fix $k \geq 0$, and suppose $\displaystyle\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$.
- **Inductive Step.** We prove true for the $k+1$ case. Consider:

  $$\sum_{i=0}^{k+1} i = (k+1) + \sum_{i=0}^{k} i$$

  By the inductive hypothesis, $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$. Now we have:

  $$\begin{aligned} &(k+1) + \sum_{i=0}^{k} i \\ &= (k+1) + \frac{k(k+1)}{2} \\ &= \frac{2(k+1) + k(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2}. \end{aligned}$$

So by the Principle of Weak Induction, the result follows. □

Remark: Notice that the inductive hypothesis was imperative in the inductive step. Once we used that $\displaystyle\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$, it was a matter of algebraic manipulation to obtain that $\displaystyle\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$. As we have verified a base case when $n = 0$ and proven that $S(k)$ implies $S(k+1)$ for an arbitrary $k \geq 0$, the Principle of Weak Induction affords us that the proposition is true.

We now examine a second example applying the Principle of Weak Induction.

Proposition 1.8. *For each $n \in \mathbb{N}$ and each $x > -1$, $(1+x)^n \geq 1 + nx$.*

Proof. The proof is by induction on n.

- **Base Case.** Consider the base case of $n = 0$. So we have $(1+x)^n = 1 \geq 1 + 0x = 1$. So the proposition holds at $n = 0$.
- **Inductive Hypothesis.** Fix $k \geq 0$ and suppose that $(1+x)^k \geq 1 + kx$.
- **Inductive Step.** We have that $(1+x)^{k+1} = (1+x)^k(1+x)$. By the inductive hypothesis, $(1+x)^k \geq (1+kx)$. So:

  $$(1+x)^k(1+x) \geq (1+kx)(1+x) = 1 + (k+1)x + kx^2 \geq 1 + (k+1)x.$$

  The last inequality follows from the fact that kx^2 is non-negative; so removing it from the right hand side will not increase that side.

So by the Principle of Weak Induction, the result follows. □

We next introduce the Principle of Strong Induction. Intuitively, strong induction is useful in proving theorems of the form "for all n, $P(n)$" where $P(k)$ alone does not neatly lend itself to forcing $P(k+1)$ to be true. Instead, it may be easier to leverage some subset of $\{P(0), \ldots, S(k)\}$ to force $P(k+1)$ to be true. Strong induction allows us to use any or all of $P(0), \ldots, P(k)$ to prove that $P(k+1)$ is true. The Principle of Strong Induction is formalized as follows.

Definition 23 (Principle of Strong Induction)**.** Let $P(n)$ be a proposition regarding an integer n, and let $k \in \mathbb{Z}$ be fixed. If:

- $P(k)$ is true; and
- for every $m \geq k$, $[P(k) \wedge P(k+1) \wedge \ldots \wedge P(m)]$ implies $P(m+1)$,

then for every $n \geq k$, the statement $P(n)$ is true.

Just as with the Principle of Weak Induction, the Principle of Strong Induction provides a format for structuring proofs.

- First, we verify for all base cases k, the statement $P(k)$ holds. This ensures that subsequent cases which rely on these early base cases are sound. For example, strong inductive proofs regarding graphs or recurrence relations may in fact have several base cases, which are used in constructing subsequent cases.
- Second, we assume that for some integer $m \geq k$, $P(k), \ldots, S(m)$ **all** hold. Notice our inductive hypothesis using strong induction assumes that **each** of the previous cases are true, while the inductive hypothesis when using weak induction only assumes $P(m)$ to be true. Strong induction assumes the extra cases because we end up using them.
- In our final step (the inductive step), we show that for an arbitrary $m \geq k$ that $P(k) \wedge P(k+1) \wedge \ldots \wedge P(m)$ implies $P(m+1)$.

Remark: The Principle of Weak Induction and the Principle of Strong Induction are equally powerful. That is, any proof using strong induction may be converted to a proof using weak induction, and vice versa. In practice, it may be easier to use strong induction, while weak induction may be clunky to use.

We illustrate how to apply the Principle of Strong Induction with a couple examples.

Proposition 1.9. *Let $f_0 = 0, f_1 = 1$; and for each natural number $n \geq 2$, let $f_n = f_{n-1} + f_{n-2}$. We have $f_n \leq 2^n$ for all $n \in \mathbb{N}$.*

Proof. The proof is by strong induction on $n \in \mathbb{N}$. Observe that $f_0 = 0 \leq 2^0 = 1$. Similarly, $f_1 = 1 \leq 2^1 = 2$. So our base cases of $n = 0, 1$ hold. Now fix $k \geq 1$; and suppose that for all $n \in \{0, \ldots, k\}$, $f_n \leq 2^n$. We now prove that $f_{k+1} \leq 2^{k+1}$. By definition of our sequence, $f_{k+1} = f_k + f_{k-1}$. We now apply the inductive hypothesis to f_k and f_{k+1}. Thus:

$$\begin{aligned} f_{k+1} &\leq 2^k + 2^{k-1} \\ &= 2^k \left(1 + \frac{1}{2}\right) \\ &\leq 2^k \cdot 2 = 2^{k+1} \end{aligned}$$

As desired. So by the Principle of Strong Induction, the result follows. □

Proposition 1.10. *Every positive integer can be written as the product of a power of 2 and an odd integer.*

Proof. The proof is by strong induction on $n \in \mathbb{Z}^+$. We have our base case $n = 1$. So $n = 1 \cdot 2^0$. Thus, the proposition holds for $n = 1$. Now fix $k \geq 1$, and suppose the proposition holds true for all $n \in [k]$. We prove true for the $k + 1$ case. We have two cases:

- Case 1: Suppose $k + 1$ is odd. Then $k + 1 = (k + 1) \cdot 2^0$, and we are done.
- Case 2: Suppose instead $k + 1$ is even. Then $k + 1 = 2h$ for some $h \in \mathbb{Z}^+$. By the inductive hypothesis, $h = m2^j$ for some odd integer m and $j \in \mathbb{N}$. Thus, $k + 1 = m2^{j+1}$, and we are done.

So by the Principle of Strong Induction, the result follows. □

1.3.1 A Brief Review of Asymptotics

The goal of this section is to provide a refresher for Big-O, Big-Omega, and Big-Theta notations, which will be of key importance in complexity theory. The purpose of asymptotics is to provide intuitive bounds on the growth rate of a function. In computer science, this function usually represents how much time or space is required to solve a problem, with respect to the input size. For this reason, we restrict attention to functions $f : \mathbb{N} \to \mathbb{N}$.

Out of the asymptotic relations, Big-O is the most common. We are also most interested in upper bounds on resource usage. How much time can this program take to run? How much space should be allocated to cover all cases? Lower bounds are important, but generally of less practical concern. We begin with the definition of Big-O.

Definition 24 (Big-O)**.** Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. We say that $f(n) \in O(g(n))$ if there exist constants $c, k \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq k$.

Given functions $f, g : \mathbb{N} \to \mathbb{N}$, there are three ways to prove that $f(n) \in O(g(n))$. The first is to use transitivity of the Big-O relation. If there exists a third function h such that $f(n) \in O(h(n))$ and $h(n) \in O(g(n))$, then we have $f(n) \in O(g(n))$. The second way is to use an induction argument. We select constants c and k, and then prove by induction that they satisfy the definition of Big-O. The third approach is to use the limit test, which we will introduce later. Let's now consider an example of the Big-O relation.

Proposition 1.11. *Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions given by $f(n) = 2^n$ and $g(n) = n!$. Then $f(n) \in O(g(n))$.*

Proof. Let $c = 1$ and $k = 4$. We show by induction that for every $n \in \mathbb{N}$ with $n \geq 4$ that $2^n \leq n!$. The base case is $n = 4$, which yields $2^4 = 16 \leq 4! = 24$. Now suppose that $2^n \leq n!$ for all $n \leq m$, where m is a fixed natural number. We prove true for the $m+1$ case. By the inductive hypothesis, we have $2^m \leq m!$. Multiplying both sides by 2 yields $2^{m+1} \leq 2 \cdot m!$. As $2 \leq 4 < m+1$, we have $2^{m+1} \leq (m+1) \cdot m! = (m+1)!$. The result follows by induction. □

We define Big-Omega similarly as for Big-O.

Definition 25 (Big-Omega)**.** Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. We say that $f(n) \in \Omega(g(n))$ if there exist constants $c, k \in \mathbb{N}$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq k$.

Remark: Note that $f(n) \in \Omega(g(n))$ if and only if $g(n) \in O(f(n))$. This fact is easy to prove by algebraic manipulations and will be left to the reader. We now define Big-Theta.

Definition 26. Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. We say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Intuitively, functions $f, g : \mathbb{N} \to \mathbb{N}$ satisfying $f(n) \in \Theta(g(n))$ grow at the same asymptotic rate. For example, $2n \in \Theta(n)$, but $2n \not\in \Theta(n^2)$ as $n^2 \not\in O(2n)$. We introduce the following test.

Theorem 1.1 (Limit Comparison Test)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. Let $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$. Then:*

- *$f(n) \in \Theta(g(n))$ if $0 < L < \infty$*
- *$f(n) \in O(g(n))$ but $f(n) \not\in \Theta(g(n))$ if $L = 0$*
- *$f(n) \in \Omega(g(n))$ but $f(n) \not\in \Theta(g(n))$ if $L = \infty$.*

Remark: Intuitively, if $L = 0$, then g grows asymptotically faster than f so we conclude $f(n) \in O(g(n))$. Similarly, if $L = \infty$, then f grows asymptotically faster than g and we conclude $f(n) \in \Omega(g(n))$. If $L = c$, then f and g grow at the same asymptotic rate and $f(n) \in \Theta(g(n))$.

Let's consider some examples.

Example 21. Let $f(n) = 3n^2 + n + 1$ and $g(n) = n^2$. Consider:

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2}{n^2} + \lim_{n\to\infty} \frac{n}{n^2} + \lim_{n\to\infty} \frac{1}{n^2} = 3 + 0 + 0 = 3$$

Therefore, since $L > 0$ is a constant, $f(n) \in \Theta(g(n))$.

Example 22. Let $f(n) = 3^n$ and $g(n) = n!$. Consider:

$$L = \lim_{n \to \infty} \frac{3^n}{n!}$$

We evaluate L using the Ratio Test for Sequences, which states that for a sequence of positive real numbers $(a_n)_{n \in \mathbb{N}}$, $\lim_{n\to\infty} \frac{a_{n+1}}{a_n} < 1$ implies that $\lim_{n\to\infty} a_n = 0$. Consider:

$$L' = \lim_{n \to \infty} \frac{3^{n+1} n!}{3^n (n+1)!} = \lim_{n \to \infty} \frac{3}{n+1} = 0$$

We conclude $L = 0$, so $3^n \in O(n!)$.

Remark: Manipulating and bounding functions is a useful skill in this class, and more importantly in an algorithms class. Series and sequence convergence tests from calculus are quite useful and worth reviewing.

1.4 Combinatorics and Graph Theory

1.4.1 Basic Enumerative Techniques

Problems from Combinatorics and Graph Theory arise frequently in computer science, especially the more theoretical areas. Combinatorics studies finite and countable discrete structures. We will focus on techniques for counting structures satisfying a given property, which is the goal of enumerative combinatorics. Computer scientists are often interested in finding optimal structures (combinatorial optimization). Many of these problems are computationally difficult. Addtionally, many enumerative problems also suffer from issues of complexity. The complexity class `#P` deals with the complexity of enumeration.

We begin with the Rule of Product and Rule of Sum. These two rules provide us the means to derive the basic combinatorial formulas, as well as enumerate more complicated sets. We need not take the Rule of Sum and Rule of Product as axioms. They can be proven quite easily. Recall the definition of a bijection (Definition 7). Two sets X and Y are defined to have the same cardinality if there exists a bijection $f : X \to Y$. The most elegant proofs of these two rules are bijective proofs. Inductive proofs also work, but are more tedious and far less enlightening. In general, inductive proofs are excellent for showing a result is true, but are rather unfulfilling. The exception to this is in graph theory. We will examine the bijective proofs for these two rules.

Before proving these results, let's discuss the intuition. The Rule of Sum allows us to count the number of elements in disjoint sets. Suppose you go to the store and have enough money for a single candy bar. If there are only five candy bars on the shelf, then you have five choices. Your choices are all disjoint- you can choose a Snickers or a Twix, but not both. Now suppose you and a friend go to the store and each have enough to purchase a single candy bar. Each of you only has five selections, and your selection of a candy bar does not affect your friend's ability to select a candy bar (and vice versa). This is the idea behind independence. The Rule of Product tells us that when two events are independent, that we multiply the number of outcomes for each to determine the number of total outcomes. In other words, there are 25 ways for you and your friend to each purchase one candy bar. We now formalize these notions.

Theorem 1.2 (Rule of Sum)**.** *Let X and Y be disjoint, finite sets, with $n = |X|$ and $m = |Y|$. Then $|X \cup Y| = |X| + |Y|$.*

Proof. (Bijective) We map $f : X \cup Y \to [n+m]$. Let $g : X \to [n]$ and $h : Y \to [m]$ be bijections. Define f as follows:

$$f(x) = \begin{cases} g(x) & : x \in X \\ h(x) + |X| & : x \in Y \end{cases} \tag{1}$$

We must show that f is a well-defined function (that it is a function and is uniquely determined for each $x \in X \cup Y$), and that it is a bijection. As X and Y are disjoint, x will be evaluated under $g(x)$ or $h(x) + |X|$, but not both. It follows from this and the fact that g and h are functions, that f is a well-defined function.

It will now be shown that f is a bijection. To show f is an injection, consider $x_1, x_2 \in X \cup Y$ such that $f(x_1) = f(x_2)$. First, observe that $f(X) = [n]$ and $f(Y) = [m+n] - [n]$. So if $f(x_1) = f(x_2)$, then

$x_1, x_2 \in X$ or $x_1, x_2 \in Y$. If $x_1, x_2 \in X$, then $f(x_1) = g(x_1)$ and $f(x_2) = g(x_2)$ and g is a bijection. Similarly, if $x_1, x_2 \in Y$, then $f(x_1) = h(x_1) + |X|$ and $f(x_2) = h(x_2) + |X|$. As h is a bijection, $x_1 = x_2$. So f is an injection.

We now show f is a surjection. As g is a surjection, each $n \in [|X|]$ has an $x \in X \cup Y$ such that $f(x) = n$. Similarly, as h is a bijection, f maps to each $n \in [n+m] - [n]$. So f is a surjection. We conclude that f is a bijection. □

Theorem 1.3 (Rule of Product)**.** *Let X and Y be sets. Then $|X \times Y| = |X| \cdot |Y|$.*

Proof. We construct a bijection $f : X \times Y \to \{0, \ldots, |X| \cdot |Y| - 1\}$. Let $g : X \to \{0, \ldots, |X| - 1\}$ and $h : Y \to \{0, \ldots, |Y| - 1\}$ be bijections. Define $f(x, y) = g(x) \cdot |Y| + h(y)$. The Division Algorithm (which we will discuss in the next section on Number Theory) guarantees that for a fixed integer q, any integer n can be written as $n = kq + r$ for some integer k and $r \in \{0, \ldots, q - 1\}$. So suppose $f(x_1, y_1) = f(x_2, y_2)$. Setting $q = |Y|$, the Division Algorithm guarantees $f(x_1, y_1) = f(x_2, y_2) = g(x_1) \cdot |Y| + h(y_1)$. Since g and h are bijections, $x_1 = x_2$ and $y_1 = y_2$. To show surjectivity, we select $n \in \{0, \ldots, |X| \cdot |Y| - 1\}$ and apply the Division Algorithm with $q = |Y|$ to obtain $n = kq + r$ for integers k, r with $0 \leq r < q$. As g, h are bijections, $x = g^{-1}(k)$ and $y = h^{-1}(r)$. So f is a surjection. We conclude f is a bijection. □

With the Rule of Product and Rule of Sum in mind, we are almost ready to begin with elementary counting techniques. The first combinatorial object of interest is the permutation. Intuitively, a permutation is a reordering of a sequence. Formally, we define it as follows.

Definition 27 (Permutation)**.** Let X be a set. A permutation is a bijection $\pi : X \to X$.

Example 23. Let $X = \{1, 2, 3\}$. The following are permutations of X: 123, 321, and 213. However, 331 is not a permutation of X since 3 is repeated twice. For the permutation 213, the function $\pi : X \to X$ maps $\pi(1) = 2, \pi(2) = 1, \pi(3) = 3$.

The natural question to ask is how many permutations exist on n elements. The answer is $n!$ permutations. We prove this formally using the Rule of Product.

Proposition 1.12. *Let X be an n-element set. There are $n!$ permutations of X.*

Proof. Without loss of generality $X = [n]$. We define a permutation $\pi : X \to X$. Observe that $\pi(1)$ can map to any of the n elements in X. As π is a bijection, only 1 can map to $\pi(1)$. This leaves $n - 1$ elements, to which $\pi(2)$ can map. The selection of $\pi(2)$ is independent of $\pi(1)$; so by Rule of Product, we multiply to obtain $n(n-1)$ ways of selecting $\pi(1)$ and $\pi(2)$. Proceeding in this manner, there are $\prod_{i=1}^{n} i = n!$ possible permutations. □

Proposition 1.13. $0! = 1$

Proof. There exists exactly one function $f : \emptyset \to \emptyset$, which is vacuously a bijection. □

Remark: When we have an n element set and want to permute $r \leq n$ elements, there are $\dfrac{n!}{(n-r)!}$ such permutations.

We now consider the word problem. We fix an alphabet, which is a finite set of characters denoted Σ. Some examples are $\Sigma = \{0, 1\}, \Sigma = \{a, b, c\}$, and the English alphabet. A word is a sequence (order matters) of characters formed from elements in Σ. Formally, a word of length n is an element $\omega \in \Sigma^n$. The Rule of Product immediately implies that there exist $|\Sigma|^n$ such words of length n. Some additional details will be provided in proof of the next proposition.

Proposition 1.14. *Let Σ be an alphabet. There are $|\Sigma|^n$ words of length n over Σ.*

Proof. We consider the n positions of $\omega \in \Sigma^n$ given by $\omega_1\omega_2\ldots\omega_n$. Each $\omega_i \in \Sigma$. The selection of each ω_i is independent of the remaining characters in ω. So by rule of product, we multiply $\prod_{i=1}^{n} |\Sigma| = |\Sigma|^n$ such words. □

Example 24. If we consider $\Sigma = \{0, 1\}$, then Proposition 1.14 implies that there exist 2^n binary strings of length n.

On the surface, the permutation and word problems may be hard to distinguish. The key difference between the two is the notion of replacement. In the permutation problem, we are given a fixed set of distinct elements. Each element is to be used precisely once in the sequence. In contrast, the word problem provides a fixed set of letters, each of which can be used for none, any, or all of the word's characters. That is, each time a letter is chosen from the alphabet, it is replaced by another instance of itsef to be used later. So 000 is a word but not a permutation because 0 is repeated.

We now discuss combination problems, which describe discrete structures where order does not matter. The simplest of these problems is the subset problem. Much like the permutation problem, the subset problem considers a set X with $|X| = n$. We seek to determine how many subsets of order k from X exist. For example, if $X = \{1, 2, 3\}$, there exist three subsets of X with two elements. These subsets are $\{1, 2\}, \{1, 3\}$, and $\{2, 3\}$.

Definition 28 (Binomial Coefficient)**.** Let X be a set with n elements. Let $k \in \mathbb{N}$. We denote the *binomial coefficient* $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$, which is read *n choose k*. We denote $\binom{X}{k}$ as the set of all k-element subsets of X. That is, $\binom{X}{k} = \{S \subseteq X : |S| = k\}$.

Proposition 1.15. *The binomial coefficient* $\binom{n}{k}$ *counts the number of k-element subsets of an n-element set.*

Proof. We begin by permuting k elements from the set, which can be done in $\frac{n!}{(n-k)!}$ ways. Define the equivalence relation $\equiv$ such that two k-element permutations σ, τ satsfy $\sigma \equiv \tau$ if and only if σ and τ are permutations of the same k elements. Each equivalence class under $\equiv$ thus has $k!$ elements, so we divide out by $k!$. This yields $\binom{n}{k}$, as desired. □

Remark: We leave it as an exercise for the reader to verify that the relation $\equiv$ in Proposition 1.15 is indeed an equivalence relation..

Example 25. Recall the example of $X = \{1, 2, 3\}$. We counted three subsets from X of order 2. Evaluating $\binom{3}{2} = 3$ confirms this result. The set $\binom{X}{3} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$.

We briefly introduce the permutation problem with repeated letters. Suppose we have a k-letter alphabet (WLOG) $[k]$, and want an n-letter permutation with a_i instances of letter i. Then there are

$$\binom{n}{a_1, a_2, a_3, \ldots, a_k} = \frac{n!}{a_1! \cdot a_2! \cdots a_k!}$$

such permutations. This is known as the multinomial coefficient. The idea is that if we have the same letter at a_i positions, we can reorder amongst those positions to obain the same permutation. So we divide $a_i!$ to account for this.

Example 26. Consider the string MISSISSIPPI. There are $\binom{11}{1,2,4,4}$ permutations of these leters.

We now discuss the multiset problem. The goal of the multiset problem is to count the ways of obtaining n objects. These objects are chosen from finite classes and are replaceable. Suppose we visit Krispy Kreme to purchase a dozen doughnuts. We can choose doughnuts from glazed, chocolate, and sprinkles. Because Krispy Kreme makes their own doughnuts, there is an infinite supply of each type. The goal is to count the number of ways we can order a dozen doughnuts. The order is the same, whether we have two chocolate doughnuts then a glazed doughnut, or the glazed doughnut between the two chocolate ones. In a sense, the multiset problem is analogous to the word problem where order does not matter. We prove below how to enumerate these objects.

Proposition 1.16 (Pipes and Dividers)**.** *Let* $n, k \in \mathbb{N}$. *There exist* $\binom{n+k-1}{n}$ *multisets with n elements each drawn from* $[k]$.

Proof. Consider the set of permutations over $n \star$ symbols and $k-1 \mid$ symbols, which we denote $\mathcal{R}(\star^n, |^{k-1})$. Let $\mathcal{M}([k])$ be the set of multisets with n elements drawn from $[k]$. We construct a bijection from $f : \mathcal{R}(\star^n, |^{k-1}) \to \mathcal{M}([k])$. Each element of $\mathcal{R}(\star^n, |^{k-1})$ is of the form $\omega = \star^{a_1} | \star^{a_2} | \ldots | \star^{a_k}$, where $\sum_{i=1}^{k} a_i = n$. We map ω to the multiset $(a_i)_{i=1}^k \in \mathcal{M}([k])$. It suffices to show this is a bijection. Consider an arbitrary multiset in $\mathcal{M}([k])$ given by the sequence $(a_i)_{i=1}^k$, where a_i denotes the number of element i in the multiset. Under f, the element $\star^{a_1}|\star^{a_2}|\ldots|\star^{a_k}$ from $\mathcal{R}(\star^n, |^{k-1})$ maps to the multiset given by $(a_i)_{i=1}^k$. So f is a surjection.

Now suppose that $f(\omega_1) = f(\omega_2)$ map to the same multiset given by the sequence $(a_i)_{i=1}^k$. By the definition of f, $\omega_1 = \omega_2 = \star^{a_1}|\star^{a_2}|\ldots|\star^{a_k}$. So f is a bijection. Recall from the discussion on the multinomial coefficient, that $|\mathcal{R}(\star^n, |^{k-1})| = \binom{n+k-1}{n,k-1} = \binom{n+k-1}{k-1} = \binom{n+k-1}{n}$. As f is a bijection, there are $\binom{n+k-1}{n}$ multisets with n elements each drawn from $[k]$. $\square$

We now consider some examples.

Example 27. Recall the doughnut problem above, where we seek to purchase a dozen doughnuts chosen from glazed, chocolate, and sprinkles. There are $\binom{12+3-1}{12}$ ways of ordering a dozen doughnuts from these options.

Example 28. Now consider a second example. Suppose there are 10 red candies and 7 blue candies, and we need to order the candies in a single-file line. We only care about the color of the candies in our order (so two red candies are treated as identical, as are two blue candies). How many single file lines are there? In order to count the single file lines, we reduce to the multiset problem. We view the blue candies as the dividers, separating multisets of red candies. Since there are 7 blue candies, we have 8 multisets. So the answer is $\binom{10+7}{10}$ single file lines.

1.4.2 Combinatorial Proofs

The goal of this section is two-fold. First, we apply the enumerative techniques from the previous section to more complicated examples. Second, we examine proofs by double counting.

Example 29. (a) How many four letter words using the alphabet $[n]$ satisfy $w_i \neq w_{i+1}$ for each $i \in [3]$? (b) How many of these words also satisfy $w_1 \neq w_4$?

(a) We select w_1 with no restrctions, in n ways. This leaves $n-1$ letters, and we choose one for w_2. The selection of w_2 is independent of w_1. By similar logic, there exist $n-1$ selections for w_3 and w_4. So by rule of product, we multiply: $n(n-1)^3$.

(b) We again select w_1 in n ways. Recall that there are $n-1$ possible selections for w_2, which are independent of the selection for w_1. We now have two cases to consider: $w_1 = w_3$ and $w_1 \neq w_3$. If $w_1 = w_3$, then there are $n-1$ selections for w_4, yielding $n(n-1)^2$ words in this case, by rule of product. If $w_1 \neq w_3$, there are $(n-2)$ possible selections for w_3, as $w_3 \neq w_2$ as well. As w_1 and w_3 are distinct, this yields $(n-2)$ possible selections for w_4. By rule of product, we multiply to obatin $n(n-1)(n-2)^2$ possible words when $w_1 \neq w_3$. The cases when $w_1 = w_3$ and $w_1 \neq w_3$ are disjoint; so by rule of sum, we add $n(n-1)^2 + n(n-1)(n-2)^2$, which counts the number of desired words.

Let us now discuss combinatorial proofs. There are two types of combinatorial proofs: bijection proofs and proofs by double counting. The notion of a bijection proof has been discussed extensively. The proof by double counting technique will now be introduced. Consider two expressions, which appear to be unrelated or in which an algebraic proof is difficult. A proof by double counting is used to show that both expressions count the same set of objects, which implies they must be equal. They have the advantage of being more elegant than algebraic proofs, as well as providing substantial insight into the problem itself.

We will prove the Binomial Theorem, which is a simple yet extremely powerful result. There is an algebraic proof by induction, which is quite ugly and unintuitive. A combinatorial proof by double counting is far more elegant.

Theorem 1.4 (Binomial Theorem)**.** $(x+y)^n = \sum_{i=0}^n \binom{n}{i} x^i y^{n-i}$

Proof. On the right hand side, there are $\binom{n}{i}$ instances of $x^i y^{n-i}$ for each $i \in \{0, \ldots, n\}$. We show the left-hand side expands to the same expression. Expanding $(x+y)^n$, each term is of the form $x^i y^{n-i}$ for some $i \in \{0, \ldots, n\}$. Each factor $(x+y)$ contributes either an x or a y (but not both) to $x^i y^{n-i}$. As multiplication commutes, the order in which the contributing factors of x are chosen does not matter. Selection of the factors contributing x fixes the selections of the factors contributing y. There are $\binom{n}{i}$ ways to select factors contributing x, so the coefficient of $x^i y^{n-i}$ is $\binom{n}{i}$. By rule of sum, we add up over all such i to obtain $\sum_{i=0}^n \binom{n}{i} x^i y^{n-i}$. $\square$

The binomial theorem and binomial identities are the source of numerous entertaining examples for combinatorial proofs. We examine a couple simple identities.

Proposition 1.17. $\sum_{i=0}^{n} \binom{n}{i} = 2^n$

Proof (Binomial Theorem). By the Binomial Theorem, $2^n = (1+1)^n = \sum_{i=0}^{n} \binom{n}{i}$. □

Proof (Double Counting). We prove by double counting. Recall that $\binom{n}{i} = |\{S \subseteq [n] : |S| = i\}|$. By rule of sum, we add up over all such $i \in \{0, \ldots, n\}$. So the left-hand side counts the number of subsets of an n-element set (WLOG $[n]$). Now consider the right hand-side. Fix a subset S. Each element $i \in [n]$ can either belong to S, or be absent from S. This choice is independent for each $i \in [n]$; so by rule of product, we multiply to obtain 2^n possible subsets of $[n]$. We conclude that $\sum_{i=0}^{n} \binom{n}{i} = 2^n$. □

Remark: There exists an explicit bijection between $2^{[n]}$ and $\{0,1\}^n$ based on the observation in the above proof for counting the right-hand side. We construct a binary string w of length n corresponding to $S \subseteq [n]$. We set $w_i = 0$ if $i \not\in S$ and $w_i = 1$ if $i \in S$. This offers a bijective proof of Proposition 1.17.

Proposition 1.18. $\sum_{i=0}^{n} \binom{n}{i} 2^i = 3^n$

Proof. The proof is by double counting. The right hand-side counts the number of strings of length n over the alphabet $[3]$. Now consider the left-hand side. Fix $i \in \{0, \ldots, n\}$. We select i slots which can be done in $\binom{n}{i}$ ways, and populate these i slots with letters from $\{0,1\}$. This fixes the remaining $n-i$ slots with the character 2. There are 2^i possible ways to populate these i slots, independent of the selection of the slots. So by rule of product, we multiply $\binom{n}{i}2^i$. By rule of sum, we add up over all such i to obtain the left-hand side. □

Proposition 1.19. $k\binom{n}{k} = n\binom{n-1}{k-1} = (n-k+1)\binom{n}{k-1}$

Proof. The proof is by double counting. We consider an n-person population, and we select a k-person committee with a president. We begin by selecting the committee members in $\binom{n}{k}$ ways, then select a seat for the president in $\binom{k}{1} = k$ ways. These selections are independent; so by rule of product, we multiply to obtain $k\binom{n}{k}$.

Now consider the term $n\binom{n-1}{k-1}$ expression. The n term counts the number of ways to select the president, while the $\binom{n-1}{k-1}$ term counts the number of ways to select the remainder of the committee from the remaining $n-1$ people. These selections are independent; so by rule of product, we multiply.

Finally, consider $(n-k+1)\binom{n}{k-1}$. The term $\binom{n}{k-1}$ counts the number of ways of selecting the committee, excluding the president. Finally, the $(n-k+1)$ term counts the number of selections of the president from the remaining $n-k+1$ people. These selections are independent; so by rule of product, we multiply. □

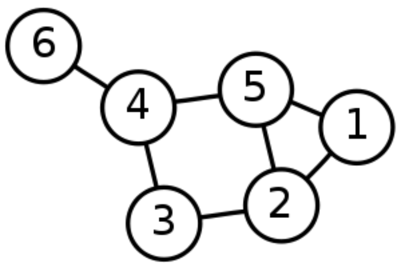
1.4.3 Graph Theory

To quote Bud Brown, "Graph theory is a subject whose deceptive simplicity masks its vast applicability." Graph theory provides simple mathematical structures known as graphs to model the relations of various objects. The applications are numerous, including efficient storage of chemicals (graph coloring), optimal assignments (matchings), distribution networks (flows), efficient storage of data (tree-based data structures), and machine learning. In automata theory, we use directed graphs to provide a visual representation of our machines. Many elementary notions from graph theory, such as path-finding and walks, come up as a result. In complexity theory, many combinatorial optimization problems of interest are graph theoretic in nature. Therefore, it is important to discuss basic notions from graph theory. We begin with the basic definition of a graph.

Definition 29 (Simple Graph)**.** A simple graph is a two-tuple $G(V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$.

By convention, a simple graph is referred to as a *graph*, and an edge $\{i, j\}$ is written as ij. In simple graphs, $ij = ji$. Two vertices i, j are said to be *adjacent* if $ij \in E(G)$. Now let's consider an example of a graph.

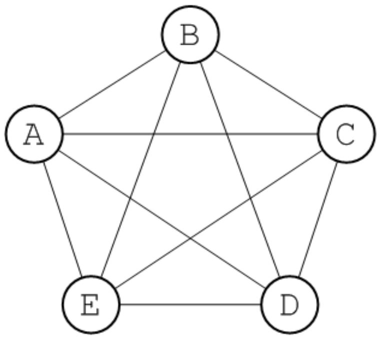
Example 30. Let $G(V, E)$ be the graph where $V = [6]$ and $E = \{12, 15, 23, 25, 34, 45, 46\}$. This graph is pictured below.



We now introduce several common classes of graphs.

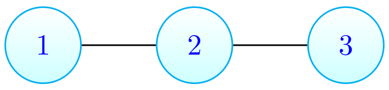
Definition 30 (Complete Graph)**.** The complete graph, denoted K_n, has the vertex set $V = [n]$ and edge set $E = \binom{V}{2}$. That is, K_n has all possible edges between vertices.

Example 31. The complete graph on five vertices K_5 is pictured below.



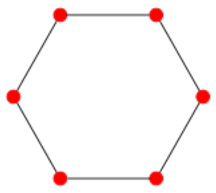
Definition 31 (Path Graph)**.** The path graph, dentoed P_n, has vertex set $V = [n]$ and the edge set $E = \{\{i, i+1\} : i \in [n-1]\}$.

Example 32. The path on three vertices P_3 is shown below.



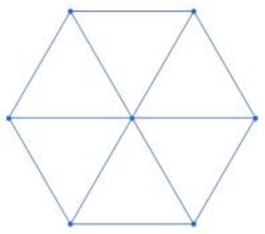
Definition 32 (Cycle Graph)**.** Let $n \geq 3$. The cycle graph, denoted C_n, has the vertex set $V = [n]$ and the edge set $E = \{\{i, i+1\} : i \in [n-1]\} \cup \{\{1, n\}\}$.

Example 33. Intuitively, C_n can be thought of as the regular n-gon. So C_3 is a triangle, C_4 is a quadrilateral, and C_5 is a pentagon. The graph C_6 is pictured below.



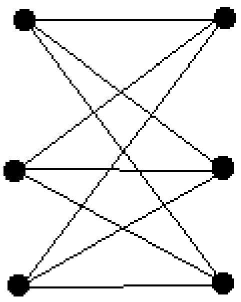
Definition 33 (Wheel Graph)**.** Let $n \geq 4$. The wheel graph, denoted W_n, is constructed by joining a vertex n to each vertex of C_{n-1}. So we take $C_{n-1} \dot{\cup} n$ and add the edges vn for each $v \in [n-1]$.

Example 34. The wheel graph on seven vertices W_7 is pictured below.



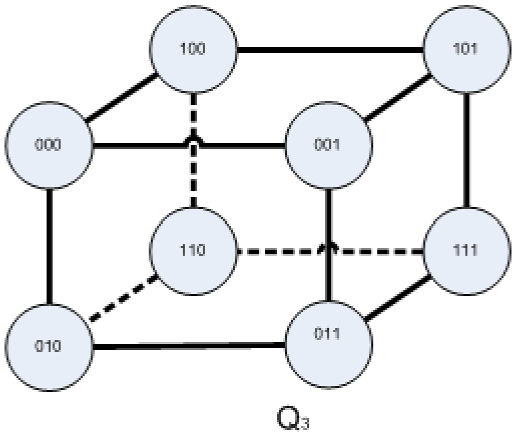
Definition 34 (Bipartite Graph)**.** A bipartite graph $G(V, E)$ has a vertex set $V = X\dot{\cup}Y$, with edge set $E \subseteq \{xy : x \in X, y \in Y\}$. That is, no two vertices in the same part of V are adjacent. So no two vertices in X are adjacent, and no two vertices in Y are adjacent.

Example 35. A common class of bipartite graphs include even-cycles C_{2n}. The complete bipartite graph is another common example. We denote the complete bipartite graph as $K_{m,n}$ which has vertex partitions $X\dot{\cup}Y$ where $|X| = m$ and $|Y| = n$. The edge set $E(K_{m,n}) = \{xy : x \in X, y \in Y\}$. The graph $K_{3,3}$ is pictured below.



Definition 35 (Hypercube)**.** The hypercube, denoted Q_n, has vertex set $V = \{0, 1\}^n$. Two vertices are adjacent if the binary strings differ in precisely one component.

Example 36. The hypercube Q_2 is isomorphic to C_4 (isomorphism roughly means that two graphs are the same, which we will formally define later). The hypercube Q_3 is pictured below.



Definition 36 (Connected Graph)**.** A graph $G(V, E)$ is said to be connected if for every $u, v \in V(G)$, there exists a $u-v$ path in G. A graph is said to be *disconnected* if it is not connected; and each connected subgraph is known as a *component.*

Example 37. So far, every graph presented has been connected. If we take two disjoint copies of any of the above graphs, their union forms a disconnected graph.

Definition 37 (Tree)**.** A Tree is a connected, acyclic graph.

Example 38. A path is an example of a tree. Additional examples include the binary search tree, the binary heap, and spanning trees of graphs. Computer science students should be familiar with all these examples.

Definition 38 (Degree)**.** Let $G(V, E)$ be a graph and let $v \in V(G)$. The degree of v, dentoed $\deg(v)$ is the number of edges containing v. That is, $\deg(v) = |\{vx : vx \in E(G)\}|$.

Example 39. Each vertex in the Cycle graph C_n has degree 2. In Example 17, $\deg(6) = 1$ and $\deg(5) = 3$.

Theorem 1.5 (Handshake Lemma)**.** *Let $G(V, E)$ be a graph. We have $\sum_{v \in V(G)} deg(v) = 2|E(G)|$.*

Proof. The proof is by double counting. The term $\deg(v)$ counts the number of edges incident to v. Each edge has two endpoints v and x, for some other $x \in V(G)$. So the edge vx is double counted in both $\deg(v)$ and $\deg(x)$. Thus, $\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. □

Remark: The Handshake Lemma is a *necessary condition* for a graph to exist. That is, all graphs satisfy the Handshake Lemma. Consider the following: does there exist a graph on 11 vertices each having degree 5? By the Handshake Lemma, $11 \cdot 5 = 2|E(G)|$. However, 55 is not even, so no such graph exists. Note that the Handshake Lemma is not a *sufficient condition.* That is, there exist degree sequences such as $(3, 3, 1, 1)$ satisfying the Handshake Lemma which are not realizable by any graph. Theorems such as Havel-Hakimi and Erdós-Gallai provide conditions that are both sufficient and necessary for a degree sequence to be realizable by some graph.

Next, the notion of a walk will be introduced. Walks on graphs come up frequently in automata theory. Intuitively, the sequence of transitions in an automaton is analogous to a walk on a graph. Additionally, algorithms like the State Reduction procedure and Brzozowski Algebraic Method that convert finite state automata to regular expressions are based on the idea of a walk on a graph.

Definition 39 (Walk)**.** Let $G(V, E)$ be a graph. A walk of length n is a sequence $(v_i)_{i=0}^{n}$ such that $v_iv_{i+1} \in E(G)$ for all $i \in \{0, \ldots, n-1\}$. If $v_0 = v_n$, the walk is said to be *closed.*

Let us develop some intuition for a walk. We start a given vertex v_0. Then we visit one of v_0's neighbors, which we call v_1. Next, we visit one of v_1's neighbors, which we call v_2. We continue this construction for the desired length of the walk. The key difference between a walk and a path is that a walk can repeat vertices, while all vertices in a path are distinct.

Example 40. Consider a walk on the hypercube Q_3. The sequence of vertices $(000, 100, 110, 111, 101)$ forms a walk, while $(000, 100, 110, 111, 101, 001, 000)$ is a closed walk. The sequence $(000, 111)$ is not a walk because 000 and 111 are not adjacent in Q_3.

We now define the adjacency matrix, which is useful for enumerating walks of a given length.

Definition 40 (Adjacency Matrix)**.** Let $G(V, E)$ be a graph. The adjacency matrix A is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & : ij \in E(G) \\ 0 & : ij \notin E(G) \end{cases} \tag{2}$$

Example 41. Consider the adjacency matrix for the graph K_5:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3}$$

Theorem 1.6. *Let $G(V, E)$ be a graph, and let A be its adjacency matrix. For each $n \in \mathbb{Z}^+$, A^n_{ij} counts the number of walks of length n starting at vertex i and ending at vertex j.*

Proof. The proof is by induction on n. When $n = 1$, we have A. By definition $A_{ij} = 1$ iff $ij \in E(G)$. All walks of length 1 correspond to the edges incident to i, so the theorem holds true when $n = 1$. Now fix $k \geq 1$ and suppose that for each $m \in [k]$ that A^m_{ij} counts the number of $i - j$ walks of length m. The $k + 1$ case will now be shown.

Consider $A^{k+1} = A^k \cdot A$ by associativity. By the inductive hypothesis, A^k_{ij} and A_{ij} count the number of $i - j$ walks of length k and 1 respectively. Observe that:

$$A^{k+1}_{ij} = \sum_{x=1}^{n} A^k_{ix} A_{xj}$$

So A^k_{ix} counts the number of ix walks of length k, and $A_{xj} = 1$ iff $xj \in E(G)$. Adding the edge xj to an $i - x$ walk of length k forms an $i - j$ walk of length $k + 1$. The result follows by induction. $\square$

We prove one more theorem before concluding with the graph theory section. In order to prove this theorem, the following lemma (or helper theorem) will be introduced first.

Lemma 1.1. *Let $G(V, E)$ be a graph. Every closed walk of odd length at least 3 in G contains an odd-cycle.*

Proof. The proof is by induction on the length of the walk. Note that a closed walk of length 3 forms a K_3. Now fix $k \geq 1$ and suppose the that any closed walk of odd length up to $2k+1$ has an odd-cycle. We prove true for walks of length $2k+3$. Let $(v_i)_{i=0}^{2k+3}$ be a walk closed of odd length. If $v_0 = v_{2k+3}$ are the only repeated vertices, then the walk itself is an odd cycle and we are done. Otherwise, suppose $v_i = v_j$ for some $0 \leq i < j \leq 2k+3$. If the walk $(v_t)_{t=i}^{k}$ is odd, then there exists an odd cycle by the inductive hypothesis. Otherwise, the walk $W = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_{2k+3})$ is of odd length at most $2k+1$. So by the inductive hypothesis, W has an odd cycle. So the lemma holds by induction. □

We now characterize bipartite graphs.

Theorem 1.7. *A graph $G(V, E)$ is bipartite if and only if it contains no cycles of odd length.*

Proof. Suppose first that G is bipartite with parts X and Y. Now consider a walk of length n. As no vertices in a fixed part are adjacent, only walks of even lengths can end back in the same part as the staring vertex. A cycle is a walk where all vertices are distinct, save for v_0 and v_n which are the same. Therefore, no cycle of odd length exists in G.

Conversely, suppose G has no cycles of odd length. We construct a bipartition of $V(G)$. Without loss of generality, suppose G is connected. For if G is not connected, we apply the same construction to each connected component. Fix the vertex v. Let $X = \{u \in V(G) : d(u, v) \text{ is even }\}$, where $d(u, v)$ denotes the distance or length of the shortest uv path. Let $Y = \{u \in V(G) : d(u, v) \text{ is odd }\}$. Clearly, $X \cap Y = \emptyset$. So it suffices to show no vertices within X are adjacent, and no vertices within Y are adjacent. Fix $v \in X$ and suppose to the contrary that two vertices in $y_1, y_2 \in Y$ are adjacent. Then there exists a closed walk of odd length $(v, \ldots, y_1, y_2, \ldots v)$. By Lemma 1.1, G must contain an odd-cycle, a contradiction. By similar argument, no vertices in X can be adjacent. So G is bipartite with bipartition $X \dot{\cup} Y$. □

1.5 Number Theory

Number Theory is one of the oldest branches of mathematics, with results dating back thousands of years. An incredibly pure and beautiful subject, Number Theory was considered useless until the 20th century with the development of public-key cryptography. In the most general sense, Number Theory studies various sets of numbers. The positive integers, particularly the prime numbers, are of particular importance. The prime numbers are the building blocks of the positive integers, which is the *Fundamental Theorem of Arithmetic.* Determining the prime factorization of a positive integer is a computationally difficult problem, though not known to be NP-Complete. Cryptography and computational number theory are active areas of research in complexity theory and quantum computing, so they are worth mentioning here. It is quite possible to make significant headway in Number Theory with three tools: the Euclidean algorithm, the Pigeonhole Principle, and the Chinese Remainder Theorem. We introduce introduce the Euclidean Algorithm and Pigeonhole Principle here.

We begin with the Pigeonhole Principle.

Theorem 1.8 (Pigeonhole Principle)**.** *Suppose we have $k \in \mathbb{Z}^+$ bins and $n > k$ objects. Then there exists a hole containing at least two objects.*

Next, we introduce the Well-Ordering Principle. This is an example of a property which seems obvious at a first glance. While it is often taken as an axiom, the Well-Ordering Principle can be proven from first principles. We do so here to illustrate how little should be taken for granted. When working at the foundations of mathematics and computing, much of what is considered obvious in conventional sets may behave differently in the general case. Clear definitions and foundational proofs lay the groundwork upon which we build a theory.

Theorem 1.9 (Well-Ordering Principle)**.** *Every non-empty $S \subseteq \mathbb{N}$ contains a minimum element.*

Proof. We first show that for each $n \in \mathbb{N}$, every non-empty subset of $\{0, \ldots, n\}$ contains a minimum element. The proof is by induction on n. When $n = 0$, the only non-empty subset is $\{0\}$. So 0 is the minimum element and we are done. Now fix $k \in \mathbb{N}$ and suppose that for every integer $0 \leq j \leq k$, every non-empty subset in $\{0, \ldots, j\}$ has a minimum element. We prove true for the $k+1$ case. Let $S \subseteq \{0, \ldots, k+1\}$. If $S = \{k+1\}$, then $k+1$ is the minimum element and we are done. Otherwise, $S \cap \{0, \ldots, k\}$ is non-empty; and so by the inductive hypothesis, $S \cap \{0, \ldots, k\}$ has a minimum element $x < k+1$, which is also the minimum element of S. So by induction, we have that for each $n \in \mathbb{N}$, every non-empty subset of $\{0, \ldots, n\}$ contains a minimum element.

We now consider an arbitrary, non-empty $S \subseteq \mathbb{N}$. Pick $x \in S$ and consider $\{0, \ldots, x\}$. By the above result, $S \cap \{0, \ldots, x\}$ is non-empty and has a minimum element, which is the minimum element of S. $\square$

Next, we introduce the notion of divisibility.

Definition 41 (Divisibility)**.** Let $a, b \in \mathbb{Z}$. We say that a divides b, denoted $a \big| b$ if there exists a $q \in \mathbb{Z}$ such that $aq = b$. If a does not divide b, we denote this $a \not| b$.

Example 42. Suppose $a = 2$ and $b = 4$. We observe that $2 \big| 4$, selecting $q = 2$. However, $3 \not| 5$.

The division algorithm is the next big result. Recall the long division procedure from grade school. This procedure gives us the integers modulo n. It also implies the Euclidean algorithm, as well as the correctness of our various number bases such as the binary number system and our standard decimal number system.

Theorem 1.10 (Division Algorithm)**.** *Let $a, b \in \mathbb{Z}$ such that $b > 0$. Then there are unique integers q and r such that $a = bq + r$ with $0 \leq r < b$. We refer to q as the quotient and r as the remainder.*

Proof. Let $Q = \{a - bq \geq 0 : q \in \mathbb{Z}\}$. If $a > 0$, $a \in Q$ (we set $q = 0$). If $a < 0$, then $a - ba \geq 0$ and so $a - ba \in Q$. So Q is non-empty. Thus, by the Well-Ordering principle, Q has a minimum. Let q^* be the associated $q \in \mathbb{Z}$ such that $a - bq \geq 0$ achieves its minimum. Let $r^* = a - bq^*$. Observe that $r^* \leq b$; otherwise, we have $a - bq^* - b \geq 0$, which implies that $a - b(q^* + 1) \geq 0$, contradicting the fact that $a - bq^*$ was the minimum element of q. It suffices to show q^* and r^* are unique.

Suppose that there exist $q_1, r_1, q_2, r_2 \in \mathbb{Z}$ such that $0 \leq r_1, r_2 < b$ and $a = bq_1 + r_1 = bq_2 + r_2$. Then $b(q_1 - q_2) = r_2 - r_1$, so $b \big| (r_2 - r_1)$. Since $-b < r_2 - r_1 < b$, it follows that $r_2 - r_1 = 0 \implies r_2 = r_1$. So $q_1 - q_2 = 0$ as well, which implies $q_1 = q_2$. And so we have uniqueness. $\square$

Example 43. Recall Example 29. We have $2 \big| 4$; which, by the division algorithm, we can write as $4 = 2(2) + 0$ setting $q = 2, r = 0$. For the example of $3 \not| 5$, we write $5 = 3(1) + 2$, setting $q = 1, r = 2$.

Number base expansion is a familiar concept from basic arithmetic. Consider the base-10 number 123. We have 3 in the ones place, 2 in the tens place, and 1 in the hundreds place. So $123 = 100 + 20 + 3 = 10^2(1) + 10^1(2) + 10^1(3)$. In fact, we can write any integer in a given number base, just like we did with 123 in base-10. The division algorithm implies this result.

Theorem 1.11. *Let $b \in \mathbb{Z}^+$ with $b > 1$. Then every positive integer n can be written uniquely in the form:*

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0$$

where $k \in \mathbb{N}$, each $a_j \in \{0, \ldots, b-1\}$, and $a_k \neq 0$.

Proof. Let $q_0, a_0 \in \mathbb{Z}$ with $0 \leq a_0 < b$ according to the Division Algorithm satisfying $n = bq_0 + a_0$. We construct a sequence of quotients and remainders as follows. For each $i > 0$, let q_i and a_i be integers with $0 \leq a_i \leq b - 1$ according to the Division Algorithm, satisfying $q_0 = bq_i + a_i$. As $b > 1$, the sequence of quotients consists of strictly decreasing positive integers. So we achieve a quotient $q_k = 0$, which is where the process terminates.

From the first equation, we have $n = bq_0 + a_0$. We substitute $q_0 = bq_1 + a_1$ to obtain $n = b(bq_1 + a_1) + a_0 = b^2 q_1 + ba_1 + a_0$. For each $i \in \{2, \ldots, n-1\}$, we substitute $q_i = bq_{i-1} + a_{i-1}$. This results in:

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0$$

We now show this sequence $(a_i)_{i=0}^k$ is unique. Suppose to the contrary that there exists a second sequence $(c_i)_{i=0}^k$ satisfying:

$$n = c_k b^k + c_{k-1} b^{k-1} + \ldots + c_1 b + c_0$$

We consider:

$$(c_k - a_k) b^k + (c_{k-1} - a_{k-1}) b^{k-1} + \ldots + (c_1 - a_1) b + (c_0 - a_0)$$

Then there exists an index j such that $a_j \neq c_j$. Without loss of generality, suppose j is the smallest such index. We factor:

$$b^j(c_k - a_k b^{k-j} + (c_{k-1} - a_{k-1})b^{k-j-1} + \ldots + (c_{j+1} - a_{j+1})b + c_j - a_j)$$

So $b^j \big| (c_j - a_j)$, which implies $b \big| (c_j - a_j)$. However, $-b < c_j - a_j < b$, so $c_j = a_j$. This implies that $c_i = a_i$ for each $i \in \{0, \ldots, k\}$. So the expansion is unique. □

The next result of interest concerns divisors. A natural question is to find common divisors between two integers. Of particular interest is the greatest common divisor and how to efficiently compute it. As it turns out, this is a computationally easy problem, taking $\mathcal{O}(\log(n))$ time to solve.

Definition 42 (Euclidean Algorithm)**.** The Euclidean Algorithm takes as input two positive integers a and b and returns their greatest common divisor. The Euclidean Algorithm works by repeatedly applying the Division Algorithm until the remainder is 0.

```
function gcd(integer a, integer b):
    if b > a:
        return gcd(b, a)
    while $b \neq 0$:
        t := b
        b := a mod b
        a := t
    return a
```

Theorem 1.12. *The Euclidean Algorithm terminates and returns the greatest common divisor of the two inputs.*

Proof. Without loss of generality, suppose $a \geq b$. Otherwise, the first iteration swaps a and b. The procedure applies the division algorithm repeatedly, resulting in a sequence of dividends, divisors, and remainders $(a_i, b_i, r_i)_{i=0}^n$ where $a_0 = a, b_0 = b$ and r_0 is the remainder from the Division Algorithm satisfying $a = bq + r_0$. For each $i > 0$, we set $a_i = b_{i-1}$, $b_i = r_{i-1}$, and r_i to satisfy the Division Algorithm: $a_i = b_i k + r_i$. As the $b_i > b_{i+1}$ for all i and each r_i satisfies $0 \leq r_i < b_i$, we have that the remainders tend to 0. Let n be the first term in which $r_i = 0$. On the next iteration, we have $b = r_i = 0$, so the algorithm terminates.

Now let $a, b \in \mathbb{Z}^+$ satisfying $a \geq b$, and let r satisfy the Division Algorithm: $a = bq + r$. It suffices to show that $\gcd(a, b) = \gcd(b, r)$, as the Euclidean algorithm computes $\gcd(r_i, r_{i+1})$ for all $i \in [n-1]$. Let d be a common divisor of a and b. Then $d \big| (a - bq)$, which implies $d \big| r$. So d is a common divisor of b and r. Now let k be a common divisor of b and r. Then we have $k \big| (bq + r)$, which implies k is a common divisor of a and b. Thus, $\gcd(a, b) = \gcd(b, r)$. □

Example 44. We use the Euclidean Algorithm to compute $\gcd(16, 6)$.

- $16 = 6(2) + 4$ $(a_0 = 16, b_0 = 6, r_0 = 4)$
- $6 = 4(1) + 2$ $(a_1 = 6, b_1 = 4, r_1 = 2)$
- $4 = 2(2) + 0$ $(a_2 = 4, b_2 = 2, r_2 = 0)$

At iteration 3, we would have $b_3 = r_2 = 0$, so the algorithm terminates. We observe $\gcd(16, 6) = 2$.

Remark: We can write $\gcd(a, b)$ as a linear combination of a and b. That is, there exist integers x, y such that $ax + by = \gcd(a, b)$. The proof is omitted, but the construction relies on backtracking from the Euclidean algorithm.

Now that we have a basic understanding of divisors, the notion of a prime number will be introduced. Prime numbers are of great theoretical interest and are incredibly important in cryptography.

Definition 43 (Prime Number)**.** An integer $p > 1$ is said to be *prime* if it is not divisible by any positive integer other than 1 and itself. An integer that is not prime is said to be *composite.*

Example 45. The numbers $2, 3, 5,$ and 7 are all prime, but 9 is composite as $3 \big| 9$.

In particular, every positive integer $n > 1$ has a prime divisor. Furthermore, there are infinitely many primes. The proofs for these two facts will be omitted here, but are standard proofs in elementary number theory. Instead, we prove the Fundamental Theorem of Arithmetic, which gives us prime factorization. In order to do so, we need two lemmas first.

Lemma 1.2. *Let $a, b, c \in \mathbb{Z}^+$ such that $gcd(a, b) = 1$ and $a \big| bc$. Then $a \big| c$.*

Proof. As $\gcd(a, b) = 1$, there exist integers x, y such that $ax + by = 1$. Multiplying both sides by c yields $axc + byc = c$. As $a \big| axc$ and $a \big| byc$ (since $a \big| bc$), it follows that $a \big| c$. □

Lemma 1.3. *Let p be a prime. If p divides $\prod_{i=1}^{n} a_i$, where each $a_i \in \mathbb{Z}^+$, then there exists a $t \in [n]$ such that $p \big| a_t$.*

Proof. The proof is by induction on n. When $n = 1$, $p \big| a_i$ and we are done. Now suppose the lemma holds true up to some $k \geq 1$. We prove true for the $k + 1$ case. Suppose $p \big| \prod_{i=1}^{n+1} a_i$, where each $a_i \in \mathbb{Z}^+$. If $p \big| \prod_{i=1}^{n} a_i$, then $p \big| a_t$ for some $t \in [n]$ by the inductive hypothesis. Otherwise, $\gcd(p, \prod_{i=1}^{n} a_i) = 1$ as p is prime. So by Lemma 1.2, $p \big| a_{n+1}$. The result follows by induction. □

Theorem 1.13 (Fundamental Theorem of Arithmetic)**.** *Every positive integer greater than* 1 *can be written uniquely as a product of primes, with the prime factors in the product written in nondecreasing order.*

Proof. It will first be shown that every positive integer greater than 1 has a prime decomposition. Suppose to the contrary that there exists a positive integer that cannot be written as the product of prime factors. By Well-Ordering, there exists a smallest such n. As n is not divisible by any primes, n itself is not prime. So $n = ab$, for $a, b \in \mathbb{Z}^+$ and $1 < a, b < n$. Because $a, b < n$, both a and b are the product of primes. So n is a product of primes, a contradiction.

We now show the uniqueness of prime decomposition by contradiction. Let m be a positive integer greater than 1 with prime decompositions $n = \prod_{i=1}^{s} p_i = \prod_{j=1}^{t} q_j$, where $p_1 \geq p_2 \geq \ldots \geq p_s$ and $q_1 \geq q_2 \geq \ldots \geq q_t$. Remove the common primes between the two factorizations and consider: $p_{i_1} p_{i_2} \ldots p_{i_u} = q_{j_1} q_{i_2} \ldots q_{i_v}$. By Lemma 1.3, each p_{i_k} divides some prime q_{j_h}, contradicting the fact the definition of a prime number. □

The congruence relation will now be introduced. The notion of congruences was introduced by Gauss at the start of the 19th century. The congruence relation enables us to process the divisibility relation in terms of equivalences. Formally, we have the following definition.

Definition 44 (Congruence Relation)**.** Let $a, b, n \in \mathbb{Z}^+$. We say that $a \equiv b \pmod{n}$ if and only if $n \big| (a - b)$.

Remark: It can easily be verified that the congruence relation $\equiv$ is an equivalence relation. This is an exercise for the reader (and a potential homework problem for my students). Note that the congruence relation is closed under addition, subtraction, and multiplication. That is, if $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then $a + c \equiv b + d \pmod{n}, a - c \equiv b - d \pmod{n}$, and $ac \equiv bd \pmod{n}$.

Intuitively, the congruence relation allows us to perform basic operations of addition and multiplication on the remainder classes after division by n. We define the integers modulo n as follows:

Definition 45 (Integers Modulo n)**.** Let $n \in \mathbb{Z}^+$. The integers modulo n, denoted $\mathbb{Z}_n$ or $\mathbb{Z}/n\mathbb{Z}$, are the set of residues or congruence classes $\{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$, where $\overline{i} \subseteq \mathbb{Z}^+$ is the set $\overline{i} = \{x \in \mathbb{Z} : x \equiv i \pmod{n}\}$.

One important question is how to efficiently calculate congruences such as $2^{9,999,999,997} \pmod{11}$. This is too large to process by hand, and it is memory intensive to try and automate these computations ignorantly. This motivates our next Finally, we prove Fermat's Little Theorem, which is of great practical as well as theoretical importance.

Theorem 1.14 (Fermat's Little Theorem)**.** *Let p be a prime and let $a \in [p - 1]$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Proof. Suppose we have an alphabet Λ with a letters. Consider the set of strings Λ^p. There exist precisely a strings consisting of the same letter repeated p times. Consider an arbitrary string $\omega = \omega_1 \omega_2 \ldots \omega_p$. The cyclic rotation of ω by j places is $\omega_j \omega_{j+1} \ldots \omega_p \omega_1 \ldots \omega_{j-1}$. Define the equivalence relation $\sim$ on Λ^p such that two strings $\omega \sim \tau$ if and only if ω is a cyclic rotation of τ. Strings consisting of a single character repeated p times each belong to an equivalence class of order 1. The remaining strings each belong to equivalence classes of order p. There are a equivalence classes of order 1, leaving $a^p - a$ remaining strings. Let k be the number of equivalence classes of order p. So $pk = a^p - a$, which is equivalent to $a^p \equiv a \pmod{p}$. □

Remark: This proof is very group-theoretic in nature, though we are hiding the group theory. More technically, we are considering the action of the cyclic group $\mathbb{Z}_p$ on Λ^p, and evaluating the orbits. We will discuss group actions in our exposition on group theory.

Example 46. Now let's use Fermat's Little Theorem to evaluate $2^{9,999,999,997} \pmod{11}$. By rule of exponents, $2^{9,999,999,997} = 2^{9,999,999,990} \cdot 2^7$. As $9,999,999,990$ is a multiple of 10, we have that $2^{9,999,999,990} = (2^{10})^{999,999,999}$ by rules of exponents. By Fermat's Little Theorem, $2^{10} \equiv 1 \pmod{11}$, which implies that $(2^{10})^{999,999,999} \equiv 1^{999,999,999} \equiv 1 \pmod{11}$. So $2^{9,999,999,990} \cdot 2^7 \equiv 2^7 \pmod{11}$. Evaluating 2^7, we see that $2^7 \equiv 7 \pmod{11}$.

1.6 Russell's Paradox and Cantor's Diagonal Argument

At the start of the 20th century, there was great interest in formalizing mathematics. In particular, the famous mathematician David Hilbert sought to show that any mathematical statement was either provably true or provably false, and that such a system was logically consistent. The mathematician Kurt Gödel showed that Hilbert's goal was unatainable with his Incompleteness Theorems, which stated that every logical system is either incomplete or inconsistent. Gödel corresponded with Church and Turing as they were developing the foundations of theoretical computer science. So many of the ideas from mathematical logic appear in theoretical computer science. The purpose of this section is to introduce Russell's Paradox, a classic set theoretic paradox, and Cantor's Diagonal Argument, which shows that $\mathbb{R}$ is uncountable. The diagonalization technique employed by Cantor comes up repeatedly in computability and complexity theory.

Definition 46 (Russell's Paradox)**.** Let $R = \{x : x \text{ is a set }, x \not\in x\}$. So R is the set of sets which do not contain themselves. Russell's Paradox asks if $R \in R$? If $R \not\in R$, then R should contain itself R contains sets which do not contain themselves. This is clearly a contradiction. Similarly, if $R \in R$, then R contains itself, contradicting the fact that it only contains sets satisfying $x \not\in x$. So $R \in R$ implies that $R \not\in R$. We obtain $R \in R$ if and only if $R \not\in R$.

Mathematical foundations, such as modern set theory, type theory, and category theory all have axioms to prohibit a Russell's Paradox scenario.

We now discuss basic countable and uncountable sets. We begin with the definition of countable.

Definition 47. A set S is said to be countable if there exists an injection $f : S \to \mathbb{N}$. Equivocally, S is countable if there exists a surjection $g : \mathbb{N} \to S$.

Clearly, every finite set is countable. We consider some infinite countable sets.

Example 47. The set $\mathbb{Z}$ is countable. Consider the injection $f : \mathbb{Z} \to \mathbb{N}$:

$$f(x) = \begin{cases} 2^x & : x \geq 0 \\ 3^{|x|} & : x < 0 \end{cases}$$

Example 48. The set $\mathbb{Q}$ is countable. Consider the injection $g : \mathbb{Q} \to \mathbb{N}$:

$$g(p/q) = \begin{cases} 2^p 3^q & : p/q > 0 \\ 0 & : p/q = 0 \\ 5^{|p|} 7^{|q|} & : p/q < 0 \end{cases}$$

Remark: These functions are injections as prime factorization is unique, which we have from the Fundamental Theorem of Arithmetic. We leave it as an exercise for the reader to verify that the functions in the previous two examples are indeed injections.

An uncountable set will now be introduced.

Theorem 1.15. *The set $2^{\mathbb{N}}$ is uncountable.*

Proof. Suppose to the contrary that $2^{\mathbb{N}}$ is countable. Let $h : \mathbb{N} \to 2^{\mathbb{N}}$ be a bijection. As h is a bijection, every element of $2^{\mathbb{N}}$ is mapped. We achieve a contradiction by constructing an element not mapped by h. Define $S \in 2^{\mathbb{N}}$ by $i \in S$ if and only if $i \not\in h(i)$. If $i \in h(i)$, then $i \not\in S$, so $h(i) \neq S$. By similar argument, if $i \not\in h(i)$, then $i \in S$, so $h(i) \neq i$. Therefore, S is not mapped by any element of $\mathbb{N}$. As our choice of h was arbitrary, no such bijection can exist. Therefore, $2^{\mathbb{N}}$ is uncountable. □

Remark: Observe that Russell's Paradox came up in the proof that $2^{\mathbb{N}}$ is uncountable. We can also use this construction to show that $\mathbb{R}$ is uncountable. More precisely, we show that $[0, 1]$ is uncountable. First, we write each $S \in 2^{\mathbb{N}}$ as an binary string $\omega_0\omega_1\ldots$ where $\omega_i = 1$ iff $i \in S$. Recall this is a bijection. So we map each infinite binary string to $0.\omega_0\omega_1\ldots \in [0, 1]$.

2 Automata Theory

Theoretical computer science is divided into three key areas: automata theory, computability theory, and complexity theory. The goal is to ascertain the power and limits of computation. In order to study these aspects, it is necessary to define precisely what constitutes a model of computation as well as what constitutes a computational problem. This is the purpose of automata theory. The computational models are automata, while the computational problems are formulated as formal languages. A common theme in theoretical computer science is the relation between computational models and the problems they solve. The Church-Turing thesis conjectures that no model of computation that is physically realizable is more powerful than the Turing Machine. In other words, the Church-Turing thesis conjectures that any problem that can be solved via computational means, can be solved by a Turing Machine. To this day, the Church-Turing thesis remains an open conjecture. For this reason, the notion of an algorithm is equated with a Turing Machine. In this section, the simplest class of automaton will be introduced- the finite state automaton, as well as the interplay with regular languages which are the computational problems finite state automata solve.

2.1 Regular Languages

In order to talk about regular languages, it is necessary to formally define a language.

Definition 48 (Alphabet)**.** An alphabet Σ is a finite set of symbols.

Example 49. Common alphabets include the binary alphabet $\{0, 1\}$, the English alphabet $\{A, B, \ldots, Z, a, b, \ldots, z\}$, and a standard deck of playing cards.

Definition 49 (Kleene Closure)**.** Let Σ be an alphabet. The Kleene closure of Σ, denoted Σ^*, is the set of all finite strings whose characters all belong to Σ. Formally, $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$. The set $\Sigma^0 = \{\epsilon\}$, where ϵ is the empty string.

Definition 50 (Language)**.** Let Σ be an alphabet. A language $L \subset \Sigma^*$.

We now delve into regular languages, starting with a definition. This definition for regular languages is rather difficult to work with and offers little intuition or insights into computation. Kleene's Theorem (which will be discussed later) provides an alternative definition for regular languages which is much more intuitive and useful for studying computation. However, the definition of a regular language provides some nice syntax for regular expressions, which are useful in pattern matching.

Definition 51 (Regular Language)**.** Let Σ be an alphabet. The following are precisely the regular languages over Σ:

- The empty language $\emptyset$ is regular.
- For each $a \in \Sigma$, $\{a\}$ is regular.
- Let L_1, L_2 be regular languages over Σ. Then $L_1 \cup L_2, L_1 \cdot L_2$, and L_1^* are all regular.

Remark: The operation $\cdot$ is string concatenation. Formally, $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$.

A regular expression is a concise algebraic description of a corresponding regular language. The algebraic formulation also provides a powerful set of tools which will be leveraged throughout the course to prove languages are regular, derive properties of regular languages, and show certain collections of regular languages are decidable. The syntax for regular expressions will now be introduced.

Definition 52 (Regular Expression)**.** Let Σ be an alphabet. A regular expression is defined as follows:

- $\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$.
- ϵ is a regular expression, with $L(\epsilon) = \{\epsilon\}$.
- For each $a \in \Sigma$, $L(a) = \{a\}$.
- Let R_1, R_2 be regular expressions. Then:
  - $R_1 + R_2$ is a regular expression, with $L(R_1 + R_2) = L(R_1) \cup L(R_2)$.
  - $R_1 R_2$ is a regular expression, with $L(R_1 R_2) = L(R_1) \cdot L(R_2)$.
  - R_1^* is a regular expression, with $L(R_1^*) = (L(R_1))^*$.

Like the definition of regular languages, the definition of regular expressions is bulky and difficult to use. We provide a couple examples of regular expressions to develop some intuition.

Example 50. Let L_1 be the set of strings over $\Sigma = \{0, 1\}$ beginning with 01. We construct the regular expression $01\Sigma^* = 01(0+1)^*$.

Example 51. Let L_2 be the set of strings over $\Sigma = \{0, 1\}$ beginning with 0 and alternating between 0 and 1. We have two cases: a string ends with 0 or it ends with 1. Suppose the string ends with 0. Then we have the regular expression $0(10)^*$. If the string ends with 1, we have the regular expression $0(10)^*1$. These two cases are disjoint, so we add them: $0(10)^* + 0(10)^*1$.

Remark: Observe in Example 51 that we are applying the Rule of Sum. Rather than counting desired objects, we are listing them explicitly. Regular Expressions behave quite similarly to the ring of integers, with several important differences, which we will discuss shortly.

Example 52. Let L_3 be the language over $\Sigma = \{a, b\}$ where the number of a's is divisible by 3. We construct a regular expression to generate L_3. We examine the cases in which there are no a's, and in which a's are present.

- **Case 1:** Suppose no a's are present. So any string consisting solely of finitely many b's belongs to L_3. Thus, we have b^* to generate these strings.
- **Case 2:** Suppose that a's are present in the string. We first construct a regular expression R to match strings that contain exactly 3 a's. We note that between two consecutive occurrences of a, there can appear finitely many b's. Similarly, there can appear finitely many b's before the first a or after the last a. So we have that: $R = b^*ab^*ab^*ab^*$. Now $R^* = (b^*ab^*ab^*ab^*)^*$ generates the set of strings in L_3 where the number of a's is divisible by 3 and at least 3 a's appear. Additionally, R^* generates ϵ, the empty string.

We note that R^* in Case 2 does not capture strings consisting solely of finitely many b's. Concatenating $b^*R^* = b^*(b^*ab^*ab^*ab^*)$ resolves this issue and is our final answer.

2.2 Finite State Automata

The finite state automaton (or FSM) is the first model of computation we shall examine. We then introduce the notion of language acceptance, culminating with Kleene's Theorem which relates regular languages to finite state automata.

We begin with the definition of a deterministic finite state automaton. There are also non-deterministic finite state automata, both with and without ϵ transitions. These two models will be introduced later. They are also equivalent to the standard deterministic finite state automaton.

Definition 53 (Finite State Automaton (Deterministic))**.** A *Deterministic Finite State Automaton* or *DFA* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite set of states,
- Σ is the alphabet,
- $\delta : Q \times \Sigma \to Q$ is the state transition function,

- q_0 is the initial state, and
- $F \subset Q$ is the set of accepting states.

We discuss how a DFA executes. Consider a string $\omega \in \Sigma^*$ as the input string for the DFA. From the initial state q_0, we transfer to another state, which we call q_1, in Q based on the first character in ω. That is, $q_1 = \delta(q_0, \omega_1)$ is the next state we visit. The second character in ω is examined and another state transition is executed based on this second character and the current state. That is, $q_2 = \delta(q_1, \omega_2)$. We repeat this for each character in the string. The state transitions are dictated by the state transition function δ associated with the machine. A string ω is said to be accepted by the finite state automaton if, when started on q_0 with ω as the input, the finite state automaton terminates on a state in F. The language of a finite state automaton M is defined as follows.

Definition 54 (Language of FSM)**.** Let M be a FSM. The language of M, denoted $L(M)$, is the set of strings that M accepts.

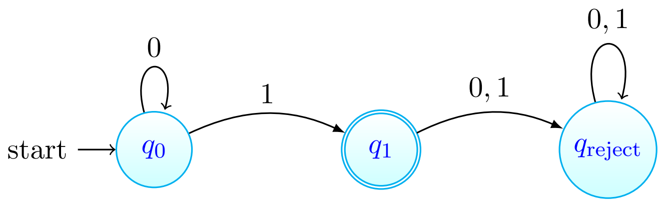
Let us consider an example of a DFA to develop some intuition.

Example 53. We design a DFA to accept the language 0^*1. Informally, we think of each state as a Boolean flag. At the initial state q_0, we simply remain at q_0 upon reading in 0's. At q_0, we transition to q_1 upon reading in a 1. This transition can be viewed as toggling a Boolean flag to indicate that the first 1 has been parsed. Now q_1 is our sole accept state.

Observe that at q_1, we have read a sequence of 0's followed by a single 1. Therefore, should the DFA read in any character at q_1, we have that the input string does **not** belong to the language generated by 0^*1. For this reason, we transition to a third state, which we call q_{reject}. At q_{reject}, the DFA simply reads in characters until it has parsed the entire string. Note that the term **reject** does not serve any functional purpose in terminating the computation early. Rather, **reject** is simply a descriptive variable name we provide to the state.

The state transition diagram for this DFA is provided below. Here, the vertices correspond to the states of the DFA, while the directed edges correspond to the transitions. Note that the loop from q_0 to itself is labeled with 0, as there is a transition $\delta(q_0, 0) = q_0$. Similarly, the directed edge from $q_0 \to q_1$ is labeled with 1, as there is a transition $\delta(q_0, 1) = q_1$. Note that we label the directed edge from $q_1 \to q_{\text{reject}}$ with both 0 and 1, as we have the transitions $\delta(q_1, 0) = q_{\text{reject}}$ and $\delta(q_1, 1) = q_{\text{reject}}$.

Finally, we note that accept states are indicated with the double circle border (as in the case of q_1), while non-accept states have the single-circle border (as in the cases of q_0 and q_{reject}).



For the sake of completeness, we identify each component of the DFA.

- The set of states $Q = \{q_0, q_1, q_{\text{reject}}\}$, where q_0 is the initial state.
- The set of accept states is $F = \{q_1\}$.
- The alphabet is $\Sigma = \{0, 1\}$.
- The transition function δ is given by the following table.

	0	1
q_0	q_0	q_1
q_1	q_{reject}	q_{reject}
q_{reject}	q_{reject}	q_{reject}

Remark: More generally, finite state automata can be represented pictorally using labeled directed graphs $G(V, E, L)$ where each state Q of the automaton is represented by a vertex of V. There is a directed edge $(q_i, q_j) \in E(G)$ if and only if there exists a transition $\delta(q_i, a) = q_j$ for some $a \in \Sigma \cup \{\epsilon\}$. The label function $L : E(G) \to 2^{\Sigma \cup \{\epsilon\}}$ maps $(q_i, q_j) \mapsto \{a \in \Sigma \cup \{\epsilon\} : \delta(q_i, a) = q_j\}$.

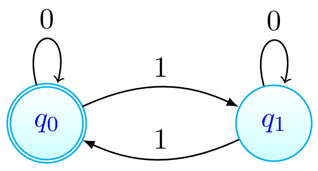
Recall from the introduction that we are moving towards the notion of an algorithm. This is actually a good starting place. Observe that a finite state automaton has no memory beyond the current state. It also has no capabilities to write to memory. Conditional statements and loops can all be reduced to state transitions, so this is a good place to start.

Consider the following algorithm to recognize binary strings with an even number of bits.

```
function evenParity(string $\omega$):
    parity := 0
    for i := 0 to len($\omega$):
        parity := (party + $\omega_i$) (mod 2)
    return parity == 0
```

So this algorithm accepts a binary string as input and examines each character. If it is a 1, then parity moves from $0 \to 1$ if it is 0, or from $1 \to 0$ if its current value is 1. So if there are an even number of 1's in ω, then parity will be 0. Otherwise, parity will be 1.

The following diagram models the algorithm as a finite state automaton. Here, we have $Q = \{q_0, q_1\}$ as our set of states with q_0 as the initial state. Observe in the algorithm above that parity only changes value when a 1 is processed. This is expressed in the finite state automata below, with the directed edges indicating that $\delta(q_i, \epsilon) = \delta(q_i, 0) = q_i, \delta(q_0, 1) = q_1$, and $\delta(q_1, 1) = q_0$. A string is accepted if and only if it has parity $= 0$, so $F = \{q_0\}$.



From this finite state automaton and algorithm above, it is relatively easy to guess that the corresponding regular expression is $(0^*10^*1)^*$. Consider 0^*10^*1. Recall that 0^* can have zero or more 0 characters. As we are starting on q_0 and $\delta(q_0, 0) = q_0, 0^*$ will leave the finite state automaton on state q_0. So then the 1 transitions the finite state automaton to state q_1. By similar analysis, the second 0^* term keeps the finite state automaton at state q_1, with the second 1 term sending the finite state automaton back to state q_0. The Kleene closure of 0^*10^*1 captures all such strings that will cause the finite state automaton to halt at the accepting state q_0.

In this case, the method of judicious guessing worked nicely. For more complicated finite state automata, there are algorithms to produce the corresponding regular expressions. We will explore one in particular, the Brzozowski Algebraic Method, later in this course. The standard algorithm in the course text is the State Reduction Method.

We briefly introduce NFAs and ϵ-NFAs prior to discussing Kleene's Theorem.

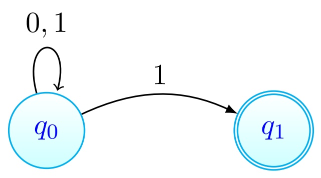
Definition 55 (Non-Deterministic Finite State Automata)**.** A *Non-Deterministic Finite State Automaton* or *NFA* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, q_0 is the initial state, and $F \subset Q$ is the set of accept states.

Remark: An ϵ-NFA is an NFA where the transition function is instead defined as $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function. An NFA is said to accept a string ω if there exists a sequence of transitions terminating in accepting state. There may be multiple accepting sequences of transitions, as well as non-accepting transitions for NFAs. Observe as well that the other difference between the non-deterministic and deterministic finite state automata is that the non-deterministic variant's transition function returns a subset of Q, while the deterministic variant's transition function returns a single state. As a result, an NFA or ϵ-NFA accepts a

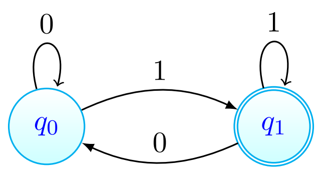
string ω precisely if there exists an accepting computation. Note that an NFA or ϵ-NFA may have multiple non-accepting computations.
With these observations in hand, we may view a DFA as an NFA, simply by restricting each transition to a single state. Similarly, an NFA is also an ϵ-NFA, where ϵ transitions are not included.

It is often easier to design efficient NFAs than DFAs. Consider an example below.

Example 54. Let L be the language given by $(0+1)^*1$. An NFA is given below. Observe that we only care about the last character being a 1. As $\delta(q_0, 1) = \{q_0, q_1\}$, the FSM is non-deterministic.



An equivalent DFA requires more thought in the design. We change state immediately upon reading a 1, then additional effort is required to ensure 1 is the last character of any valid string. At q_1, we transition to q_0 upon reading a 0, as that does not guarantee 1 is the last character of the string. If at q_1, we remain there upon reading in additional 1's.



We introduce one final definition before Kleene's Theorem, namely a *complete computation*. Intuitively, a complete computation is the sequence of states that the given finite state automaton visits when parsing a given input string ω. This is formalized as follows.

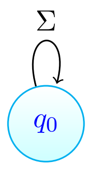
Definition 56 (Complete Computation)**.** Let M be a finite state automaton, and let $\omega \in \Sigma^*$ be of length n. A *complete computation* of M on ω is a sequence of states $(s_i)_{i=1}^n$ where $s_0 = q_0$; and for each $i \in \{0, \ldots, n-1\}$, $s_{i+1} \in \delta(s_i, \omega_i)$. To allow for ϵ transitions, we allow that each $\omega_i \in \Sigma \cup \{\epsilon\}$. The complete computation is *accepting* if and only if $s_n \in F$. That is, the computation is *accepting* if and only if M halts on an accept state when run on ω. The computation is said to be *rejecting* otherwise.

We now conclude this section with Kleene's Theorem, for which we provide a proof sketch.

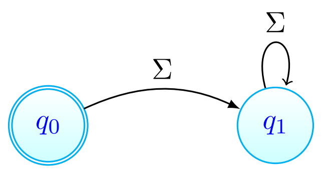
Theorem 2.1 (Kleene)**.** *A language L is regular if and only if it is accepted by some DFA.*

Proof Sketch. We first show that any regular language L is accepted by a DFA. The proof is by induction on $|L|$. When $L = 0$, a DFA with no accept states accepts L. Now suppose $|L| = 1$. There are two cases to consider: $L = \{\epsilon\}$ and $L = \{a\}$ for some $a \in \Sigma$. Suppose first $L = \{\epsilon\}$. We define a two-state DFA M_ϵ where $F = \{q_0\}$, and we transition from q_0 to q_1 upon reading in any character from Σ; after which, we remain at q_1.

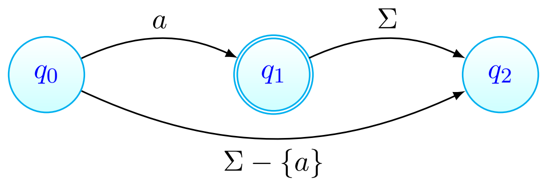
Now suppose $L = \{a\}$. We define a three-state DFA as follows with $F = \{q_1\}$. We have $\delta(q_0, a) = q_1$ and $\delta(q_1, x) = q_2$ for any $x \in \Sigma$. Now for any $y \in \Sigma - \{a\}$, we have $\delta(q_0, y) = q_2$.



A DFA to accept $L = \emptyset$.



A DFA to accept $L = \{\epsilon\}$.



A DFA to accept $L = \{a\}$, for some $a \in \Sigma$.

Now fix $n \in \mathbb{N}$ and suppose that for any regular language L with cardinality at most n, that L is accepted by some DFA. Let L_1, L_2 be regular languages with cardinalities at most n. We show that $L_1 \cup L_2, L_1L_2$, and L_1^* are all accepted by some DFA.

Lemma 2.1. *Let L_1, L_2 be regular languages accepted by DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$ respectively. Then $L_1 \cup L_2$ is accepted by some DFA.*

There are two proofs of Lemma 2.1. The first involves constructing an ϵ-NFA, which is quite intuitive and also useful in a later procedure to convert regular expressions to finite state automata. We simply add a new start state, with ϵ transitions to the initial states of M_1 and M_2. We leave the details of this proof as an exercise for the reader.

The second proof proceeds by running M_1 and M_2 in parallel. The idea is that we track the current states of M_1 and M_2. Each time a character is read, we run δ_1 on the current state of M_1 and run δ_2 on the current state of M_2. Now the input string ω is accepted if and only if M_1 or M_2 (or both) accepts ω. We formalize this with a *product machine* M.

Definition 57 (Product Machine)**.** Let M_1 and M_2 be finite state automata. A *product machine* M is a finite state automaton defined as follows.

- The state set of M is $Q_1 \times Q_2$, which allows us to track the current states of both M_1 and M_2 as we run these machines in parallel.
- The transition function of M, $\delta_M = \delta_1 \times \delta_2$, which formalizes the notion of running δ_1 on the current state of M_1 and δ_2 on the current state of M_2.
- The initial state of M is (q_{0_1}, q_{0_2}), the ordered pair consisting of the initial states of M_1 and M_2.
- The alphabet of M is $\Sigma = \Sigma(M_1) \cup \Sigma(M_2)$. Though in practice, M_1 and M_2 usually have the same alphabet.
- As with finite state automata in general, the set of final states for M is simply a subset of its state set $Q_1 \times Q_2$. There are no other constraints on the set of final states. These may (and should) be chosen strategically when constructing M.

Remark: For the proof of Lemma 2.1, our goal is to construct a product machine M from M_1 and M_2 to accept $L_1 \cup L_2$. So M_1 has to end in an accept state or M_2 has to end in an accept state. Thus, the accept states of M are $(Q_1 \times F_2) \cup (F_1 \times Q_2)$.

Proof of Lemma 2.1. We construct a DFA to accept $L_1 \cup L_2$. Let M be such a DFA, with $Q(M) = Q_1 \times Q_2$, $\Sigma(M) = \Sigma$, $\delta_M = \delta_1 \times \delta_2$, $q_0(M) = (q_{0_1}, q_{0_2})$, and $F(M) = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. It suffices to show that $L(M) = L_1 \cup L_2$.

Let $\omega \in L(M)$ be a string of length n. Then there is a complete accepting computation $\hat{\delta_M}(\omega) \in Q(M)$. Let (q_{a_n}, q_{b_n}) be the final state in $\hat{\delta_M}(\omega)$. If $q_{a_n} \in F_1$, then the projection of $\hat{\delta_M}$ into the first component is an accepting computation of M_1, so $\omega \in L_1$. Otherwise, $q_{b_n} \in F_2$ and the projection of $\hat{\delta_M}$ into the second component is an accepting computation of M_2. So $\omega \in L_2$.

Let $\omega \in L_1 \cup L_2$. Let $\hat{\delta_{M_1}}(\omega)$ and $\hat{\delta_{M_2}}(\omega)$ be complete computations of M_1 and M_2 respectively. One of these computations must be accepting, so $\hat{\delta_{M_1}} \times \hat{\delta_{M_2}}$ is an accepting complete computation of M. Thus, $\omega \in L(M)$. So $L(M) = L_1 \cup L_2$. □

Lemma 2.2. *Let L_1, L_2 be regular languages accepted by DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ respectively. Then L_1L_2 is accepted by some NFA.*

Proof. We construct an ϵ-NFA M to accept L_1L_2 as follows. Let $Q_M = Q_1 \cup Q_2$, $\Sigma_M = \Sigma$, $q_{0M} = q_{01}$, and $F_M = F_2$. We now construct $\delta_M = \delta_1 \cup \delta_2 \cup \{((q_i, \epsilon), q_{02}) : q_i \in F_1\}$. That is, we add an ϵ transition from each state of F_1 to the initial state of M_2. It suffices to show that $L(M) = L_1L_2$.

Let $\omega \in L(M)$. Then there exists an accepting complete computation $\hat{\delta_M}(\omega)$. By construction of M, $\hat{\delta_M}(\omega)$ contains some state $q_i \in F_1$ followed by q_{02}. So the string $\omega_1 \ldots \omega_{i-1} \in L_1$ and $\omega_{i+1} \ldots \omega_{|\omega|} \in L_2$. Conversely, let $x \in L_1, y \in L_2$, and let $\hat{\delta_{M_1}}(x)$ and $\hat{\delta_{M_2}}(y)$ be accepting complete computations of M_1 on x and M_2 on y respectively. Then $\hat{\delta_{M_1}}\hat{\delta_{M_2}}$ is a complete accepting computation of M_2, as $q_{|x|} \in \hat{\delta_{M_1}}$ has an ϵ-transition to q_{02} under δ_M. So $xy \in L(M)$. Thus, $L(M) = L_1L_2$. □

Lemma 2.3. *Let L be a regular language accepted by a FSM $M = (Q, \Sigma, \delta, q_0, F)$. Then L^* is accepted by some FSM.*

Proof. We construct an ϵ-NFA M^* to accept L^*. We modify M as follows to obtain M^*. Set $F_{M^*} = F_M \cup \{q_0\}$, and set $\delta_{M^*} = \delta_M \cup \{((q_i, \epsilon), q_0) : q_i \in F_M\}$. It suffices to show $L(M^*) = L^*$. Suppose $\omega \in L(M^*)$. Let $\hat{\delta_{M^*}}(\omega)$ be an accepting complete computation. Let $(a_i)_{i=1}^{k}$ be the indices in which q_0 is visited from an accepting state. Then for each $i \in [k-1]$, $\omega_{a_i} \ldots \omega_{a_{i+1}-1} \in L$. So $\omega \in L^*$.

Conversely, suppose $\omega = \omega_1\omega_2\omega_3 \ldots \omega_k \in L^*$. For each $i \in [k]$, let $\hat{\delta_M}(\omega_i)$ be an accepting complete computation. As there is an ϵ transition from each state in F to q_0 in M^*, the concatenation $\prod_{i=1}^{k} \hat{\delta_M}(\omega_i)$ is an accepting complete computation of M^*. So $\omega \in L(M^*)$. □

To complete the forward direction of the proof, it is necessary to show that ϵ-NFAs, NFAs, and DFAs are equally powerful. This will be shown in a subsequent section. In order to prove the converse, it suffices to exhibit an algorithm to convert a DFA to a regular expression, then argue the correctness of the algorithm. To this end, we present the Brzozowski Algebraic Method in a subsequent section. □

2.3 Converting from Regular Expressions to ϵ-NFA

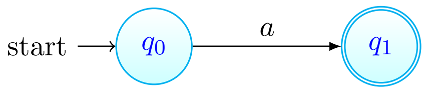
Regular expressions are important from a theoretical standpoint, in providing a concise description of regular languages. They are also of practical importance in pattern matching, with various programming languages providing regular expression libraries. These libraries construct FSMs from the regular expressions to validate input strings. One such algorithm is Thompson's Construction Algorithm. Formally:

Definition 58 (Thompson's Construction Algorithm)**.**

- `Instance:` A regular expression R.
- `Output:` An ϵ-NFA N with precisely one final state such that $L(N) = L(R)$.

The algorithm is defined recursively as follows:

- Suppose $R = \emptyset$. Then we return a singe state FSM, which does not accept any string.
- Suppose $R = a$, where $a \in \Sigma \cup \{a\}$. We define a two-state machine as shown below:



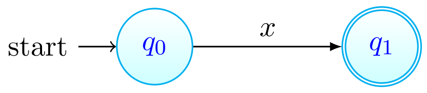
- Suppose $R = P + S$, where P and S are regular expressions. Let M_P and M_S be the ϵ-NFAs accepting P and S respectively, by applying Thompson's Construction Algorithm to P and S respectively. We define an ϵ-NFA to accept R as follows. We add an initial state q_R and the transitions $\delta(q_R, \epsilon) = \{q_P, q_S\}$, the initial states of R and S respectively. As M_R and M_S were obtained from Thompson's Construction Algorithm, they each have a single final state. We now add a new state q_{F_R} and transitions $\delta(q_{F_P}, \epsilon) = \delta(q_{F_P}, \epsilon) = \{q_{F_R}\}$, and set $F_R = \{q_{F_R}\}$.

- Suppose $R = PS$, where P and S are regular expressions. Let M_P and M_S be the ϵ-NFAs accepting P and S respectively, by applying Thompson's Construction Algorithm to P and S respectively. We construct an ϵ-NFA M_R to accept R. We begin by setting the final state of M_P is the initial state of M_S. Then $F_R = F_S$.

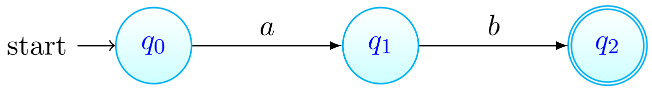
- Now consider R^*. Let M_R be the ϵ-NFA accepting R, by applying Thompson's Construction Algorithm to R. We construct an ϵ-NFA to accept R^* as follows: We add a new state initial state q_{R^*} and a new final state $q_{F_{R^*}}$. We then add the transitions $\delta(q_{R^*}, \epsilon) = \{q_R, q_{F_{R^*}}\}$ and $\delta(q_{F_R}, \epsilon) = \{q_R\}$.

Thompson's Construction Algorithm follows immediately from Kleene's Theorem. We actually could use the constructions given in this algorithm for the closure properties of union, concatenation, and Kleene closure in Kleene's Theorem. This is a case where a proof gives us an algorithm. As a result, we omit a formal proof of Thompson's Construction Algorithm, and we proceed with an example to illustrate the concept.

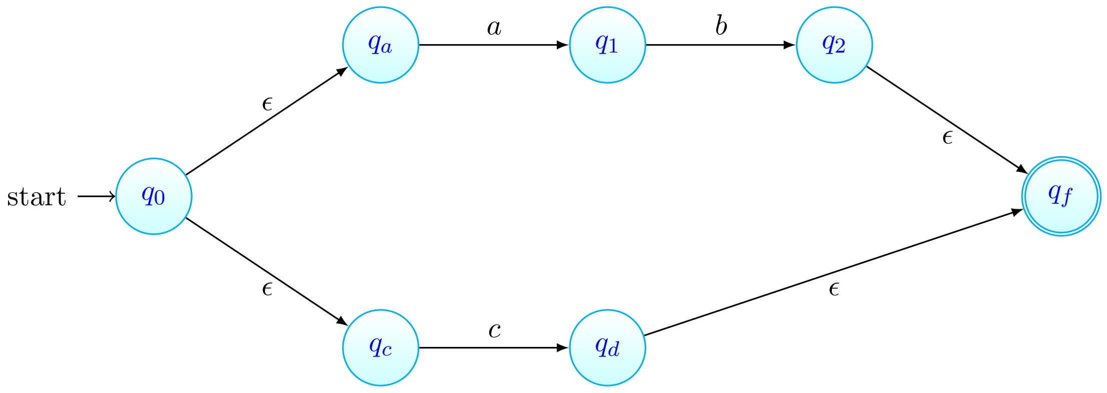
Example 55. Let $R = (ab + c)^*$ be a regular expression. We construct an ϵ-NFA recognizing R using Thompson's Construction Algorithm. Observe that our base cases are the regular expressions a, b and c. So for each $x \in \{a, b, c\}$, we construct the FSMs:



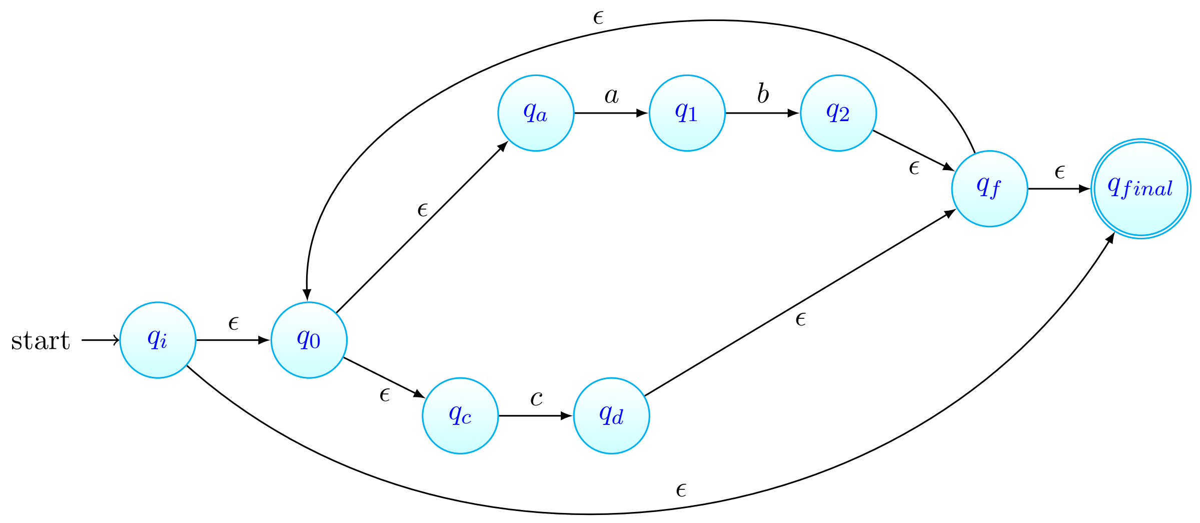
Now we apply the concatenation rule for ab to obtain:



Next, we apply the union rule for $ab + c$ to obtain:



Finally, we apply the Kleene closure step to $(ab + c)^*$ to obtain our final ϵ-NFA:



2.4 Algebraic Structure of Regular Languages

Understanding the algebraic structure of regular languages provides deep insights; which from a practical perspective, allow for the design of simpler regular expressions and finite state automata. Leveraging these machines also provides elegant and useful results in deciding certain collections of regular languages, which will be discussed in greater depth when we hit computability theory. Intuitively, the set of regular languages over the alphabet Σ has a very similar algebraic structure to the integers. This immediately translates into manipulating regular expressions, applying techniques such as factoring and distribution. We begin with the definition of a group, then continue on to other algebraic structures such as semi-groups, monoids, rings, and semi-rings. Ultimately, the algebra presented in this section will be subservient to deepening our understanding of regular languages. In a later section, the exposition of group theory will be broadened to include the basics of homomorphisms and group actions.

Definition 59 (Group)**.** A *Group* is a set of elements G with a closed binary operator $\star : G \times G \to G$ satisfying the following axioms:

1. Associativity: For every $g, h, k \in G$, $(g \star h) \star k = g \star (h \star k)$
2. Identity: There exists an element $1 \in G$ such that $1g = g1 = g$ for every $g \in G$.
3. Inverse: For every g, there exists a g^{-1} such that $gg^{-1} = g^{-1}g = 1$.

Remark: By convention, we drop the $\star$ operator and write $g \star h$ as gh, for a group is an abstraction over the operation of multiplication. When $\star$ is commutative, we write $g \star h$ as $g + h$ (explicitly using the $+$ symbol), with the identity labeled as 0. This is a convention which carries over to ring and field theory.

Example 56. The set of integers $\mathbb{Z}$ forms a group over addition. However, $\mathbb{Z}$ with the operation of multiplication fails to form a group.

Example 57. The real numbers $\mathbb{R}$ form a group over addition, and $\mathbb{R} - \{0\}$ forms a group over multiplication.

Example 58. The integers modulo $n \geq 1$, denoted $\mathbb{Z}_n$ or $\mathbb{Z}/n\mathbb{Z}$, forms a group over addition, and $\mathbb{Z}/n\mathbb{Z} - \{\overline{0}\}$ forms a group over multiplication precisely when n is a prime.

We defer formal proofs that these sets form groups under the given operations, until our group theory unit. The purpose of this section is purely intuitive. The next structure we introduce is a ring.

Definition 60 (Ring)**.** A ring is a three-tuple $(R, +, *)$, where $(R, +)$ forms an Abelian group, and $* : R \times R \to R$ is closed and associative. Additionally, multiplication distributes over addition: $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$ for all $a, b, c \in R$.

Remark: A ring with a commutative multiplication is known as a *commutative ring*, and a ring with a multiplicative identity 1 is known as a *ring with unity.* If $R - \{0\}$ forms an Abelian group over the operation of multiplication, then R is known as a field. Each of the above groups forms a ring over the normal operation of multiplication. However, only $\mathbb{R}, \mathbb{Q}, \mathbb{C}$ and $\mathbb{Z}/p\mathbb{Z}$ (for p prime) are fields.

We now have some basic intuition about some common algeraic structures. Mathematicians focus heavily on groups, rings, and fields. Computer scientists tend to make greater use of monoids, semigroups, and posets (partially ordered sets). The set of regular languages over the alphabet Σ forms a semi-ring, which is a monoid over the addition operation (set union) and a semigroup over the multiplication operation (string concatenation). We formally define a monoid and semigroup, then proceed to discuss some intuitive relations between the semi-ring of regular languages and the ring of integers.

Definition 61 (Semigroup)**.** A *semigroup* is a two-tuple $(S, \odot)$ where $\odot$ is a closed, binary operator $\odot : S \times S \to S$.

Definition 62 (Monoid)**.** A *monoid* is a two-tuple $(M, \odot)$ that forms a semigroup, with identity.

Remark: A monoid with inverses is a group.

Definition 63 (Semi-Ring)**.** A *semi-ring* is a three-tuple $(R, +, *)$ is a commutative monoid over addition and a semigroup over multiplication. Furthermore, multiplication distributes over addition. That is, $a(b+c) = ab+ac$ and $(b+c)a = ba + ca$ for all $a, b, c \in R$.

Recall the proof of the binomial theorem. We had a product $(x+y)^n$, and selected k of the factors to contribute an x term. This fixed the remaining $n-k$ terms to contribute a y, yielding the term $\binom{n}{k}x^k y^{n-k}$. Prior to rearranging and grouping common terms, the expansion yields strings of length n consisting solely of characters drawn from $\{x, y\}$. So the ith $(x+y)$ factor contributes either x or y (but not both) to character i in the string, just as with a regular expression. Each selection is independent; so by rule of product, we multiply. Since $\mathbb{Z}$ is a commutative ring, we can rearrange and group common terms. However, string concatenation is a non-commutative multiplication, so we cannot rearrange. However, the rule of sum and rule of product are clearly expressed in the regular expression algebra.

Example 59. While commutativity of multiplication is one noticeable difference between the integers and regular languages, factoring and distribution remain the same. Recall the regular expression from Example 38, $0(10)^*0 + 0(10)^*1$. We can factor $0(10)^*$ to achieve an equivalent regular expression $0(10)^*(\epsilon + 1)$.

Example 60. We construct a regular expression over $\Sigma = \{a, b, c\}$, where $n_b(\omega) + n_c(\omega) = 3$ (where $n_b(\omega)$ denotes the number of b's in ω), using exactly one term. We note that there can be arbitrarily many a's between each pair of consecutive b/c's. So we start with a^*, then select either a b or c. This is formalized by $a^*(b+c)$. We then repeat this logic twice more to obtain:

$$a^*(b+c)a^*(b+c)a^*(b+c)a^*.$$

Notice how much cleaner this answer is than constructing regular expressions for all 8 cases where $n_b(\omega) + n_c(\omega) = 3$.

2.5 DFAs, NFAs, and ϵ-NFAs

In this section, we show that DFAs, NFAs, and ϵ-NFAs are equally powerful. We know already that DFAs are no more powerful than NFAs, and that NFAs are no more powerful than ϵ-NFAs. The approach is to take the machine with weakly greater power and convert it to an equivalent machine of weakly less power. Note my use of weak ordering here. We begin with a procedure to convert NFAs to DFAs.

Recall that an NFA may have multiple complete computations for any given input string ω. Some, all, or none of these complete computations may be accepting. In order for the NFA to accept ω, at least one such complete computation must be accepting. The idea in converting an NFA to a DFA is to enumerate the possible computations for a given string. The power set of Q_{NFA} becomes the set of states for the DFA. In the NFA, a given state is selected non-deterministically in a transition. The DFA deterministically selects all possible states.

Definition 64 (NFA to DFA Algorithm)**.** Formally, we define the input and output.

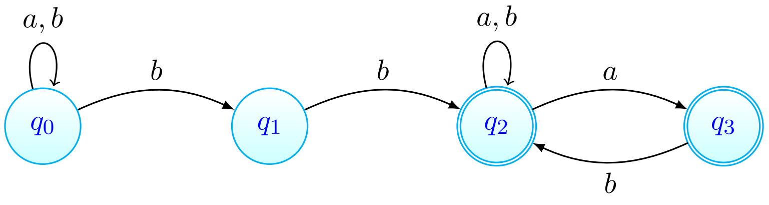
- `Instance:` An NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Note that N is **not** an ϵ-NFA.
- `Output:` A DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D\}$ such that $L(D) = L(N)$.

We construct D as follows:

- $Q_D = 2^{Q_N}$
- For each $S \in 2^{Q_N}$ and character $x \in \Sigma$, $\delta_D(S, x) = \bigcup_{s \in S} \delta_N(S, x)$
- $q_D = \{q_0\}$
- $F_D = \{S \in 2^{Q_N} : S \cap F_N \neq \emptyset\}$

Consider an example:

Example 61. We seek to convert the following NFA to a DFA.



The most methodical way to represent the DFA is by providing the transition table, noting the initial and accept states. We use the $\rightarrow$ symbol to denote the start state and the $\star$ symbol to denote accept states. The empty set is not included, because no transition leaves this state. Intuitively, the empty set is considered a trap state.

State	**a**	**b**
$\rightarrow \{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_1, q_2\}$
$\star\{q_0, q_2\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2\}$
$\star\{q_0, q_3\}$	$\{q_0\}$	$\{q_0, q_1, q_2\}$
$\star\{q_1, q_2\}$	$\{q_2, q_3\}$	$\{q_2\}$
$\star\{q_1, q_3\}$	$\{q_0\}$	$\{q_0, q_1, q_2\}$
$\star\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2\}$
$\star\{q_0, q_1, q_2\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2\}$
$\star\{q_0, q_1, q_3\}$	$\{q_0\}$	$\{q_0, q_1, q_2\}$
$\star\{q_0, q_2, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2\}$
$\star\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_2\}$
$\star\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_2\}$

Dealing with an exponential number of states is tedious and bothersome. We can prune unreachable states after constructing the transition table, or we can only add states as they become necessary. In Example 44, only the states $\{q_0\}, \{q_0, q_1\}, \{q_0, q_1, q_2\}$, and $\{q_0, q_2, q_3\}$ are reachable from q_0. So we may restrict attention to those. A graph theory intuition regarding connected components and walks is useful here. If the graph is not connected, no path (and therefore, walk) exists between two vertices on separate components. We represent our FSMs pictorally as graphs, which allows us to easily apply the graph theory intuition.

We now prove the correctness of this algorithm.

Theorem 2.2. *Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA. There exists a DFA D such that $L(N) = L(D)$.*

Proof. Let D be the DFA returned from the algorithm in Definition 50. It suffices to show that $L(N) = L(D)$. Let $\omega \in L(N)$. Then there is a complete accepting complete computation $\hat{\delta_N}(\omega) = (q_0, \ldots, q_k)$. Let $\hat{\delta_D}(\omega) = (p_0, \ldots, p_k)$ be the complete computation of D on ω. We show by induction that $q_i \in p_i$ for all $i \in \{0, \ldots, k\}$.

When $i = 0$, $p_0 = \{q_0\}$, so the claim holds. Now fix h such that $0 < h < k$ and suppose that for each $i < h$, $q_i \in p_i$. We prove true for the $h + 1$ case. By construction:

$$p_{h+1} = \bigcup_{q \in p_h} \delta_N(q, \omega_{h+1}) \tag{4}$$

Since $q_i \in p_h$ and $q_{i+1} \in \delta_N(q_i, \omega_{h+1})$, we have $q_{i+1} \in \delta_D(q_i, \omega_{h+1})$. Since $\omega \in L(N)$, $q_k \in F_N$. We know that $q_k \in p_k$. By construction of D, $p_k \in F_D$. So $\omega \in L(D)$, which implies $L(N) \subset L(D)$.

Conversely, suppose $\omega \in L(D)$. Let $\hat{\delta_D}(\omega) = (p_1, \ldots, p_k)$ be an accepting complete computation of D on ω. We construct an accepting complete computation of N on ω. As $\omega \in L(D)$, $p_k \cap F_n \neq \emptyset$. Let $q_f \in p_k \cap F_n$. From (7), $q_f \in \delta_N(q, \omega_{k-1})$ for some $q \in p_{k-1}$. Let q_{k-1} be such a state. Iterating on this argument yields a sequence of states starting at q_0 and ending at q_f, which is a complete accepting computation of N on ω. So $\omega \in L(N)$, which implies $L(D) \subset L(N)$. □

Example 62. In the worst case, the algorithm in Definition 50 requires an exponential number of cases. Consider an n state NFA where $\delta(q_0, 0) = \{q_0\}$, $\delta(q_0, 1) = \{q_0, q_1\}$; and for all $i \in [n-1]$, $\delta(q_i, 0) = \delta(q_i, 1) = \{q_{i+1}\}$. The only accept state is q_n.

We now discuss the procedure to convert an ϵ-NFA to an NFA without ϵ transitions. This shows that an ϵ-NFA is no more powerful than an NFA. We know already that an NFA is no more powerful than an ϵ-NFA, so our construction shows that an NFA and ϵ-NFA are equally powerful. The definition of the ϵ-closure will first be introduced.

Definition 65 (ϵ-Closure)**.** Let N be an ϵ-NFA and let $q \in Q$. The ϵ-closure of q, denoted ECLOSE(q) is defined recursively as follows. First, $q \in$ ECLOSE(q). Next, if the state $s \in$ ECLOSE(q) and there exists a state r such that $r \in \delta(s, \epsilon)$, then $r \in$ ECLOSE(q).

Remark: Intuitively, ECLOSE(q) is the set of states reachable from q using only ϵ transitions.

Definition 66 (ϵ-NFA to NFA Algorithm)**.** We begin with the instance and output statements:

- `Instance`: An ϵ-NFA $N = (Q_N, \Sigma, \delta, q_{0_N}, F)$.
- `Output`: An NFA without ϵ transitions $M = (Q_M, \Sigma, \delta, q_{0_M}, F)$ such that $L(M) = L(N)$.

We construct M as follows:

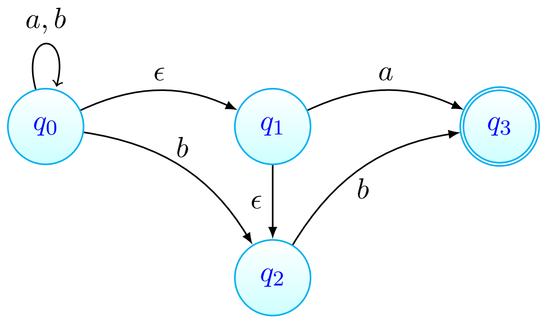
```
if $N$ has no $\epsilon$ transitions:
    return $N$

for each $q \in Q_N$:
    compute ECLOSE($q$)
    for each pair $s \in$ ECLOSE($q$):
        if $s \in F_N$:
            $F_N := F_N \cup \{q\}$

        for each $\omega \in \Sigma$:
            if $\delta(s, \omega)$ is defined:
                add transition $\delta(q, \omega) := \delta(s, \omega)$
                remove $s$ from $\delta(q, \epsilon)$
return the modified $N$
```

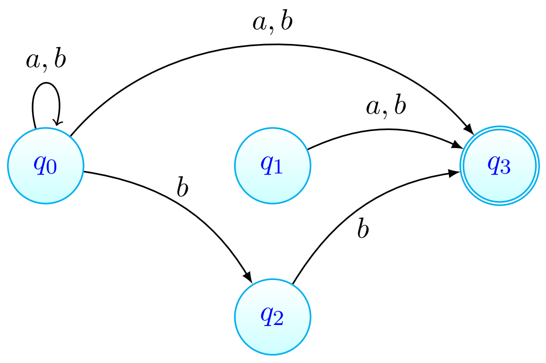
Theorem 2.3. *The procedure in Definition 66 correctly constructs an NFA M with no ϵ transitions such that $L(M) = L(N)$.*

Example 63. Consider the following ϵ-NFA:



We begin by computing the ϵ-closure of each state. The only states with ϵ-transitions are q_0 and q_1, so we restrict attention to the ϵ-closures for these states. We have $\text{ECLOSE}(q_1) = \{q_1, q_2\}$ and $\text{ECLOSE}(q_0) = \{q_0\} \cup \text{ECLOSE}(q_1) = \{q_0, q_1, q_2\}$. Since $\delta(q_2, b) = q_3$, we add the transitions $\delta(q_0, b) = \delta(q_1, b) = q_3$ and remove the ϵ transition $\delta(q_1, \epsilon) = q_2$.

We repeat this procedure for q_0. Since $\delta(q_1, a) = \delta(q_1, b) = q_3$. So we add the transitions $\delta(q_0, a) = \delta(q_0, b) = q_3$ and remove the transition $\delta(q_0, \epsilon) = q_1$. The final NFA is as follows:



2.6 DFAs to Regular Expressions- Brzozowski's Algebraic Method

In order to complete the proof of Kleene's Theorem, we must show that each finite state machine accepts a regular language. To do so, it suffices to provide an algorithm that converts a FSM to a corresponding regular expression. We exmaine the Brzozowski algebraic method. Intuitively, the Brzozowski Algebraic Method takes a finite state automata diagram (the directed graph) and constructs a system of linear equations to solve. Solving a subset of these equations will yield the regular expression for the finite state automata. I begin by defining some notation. Fix a FSM M. Let E_i denote the regular expression such that $L(E_i)$ contains strings ω such that when we run M on ω, it halts on q_i.

The system of equations consists of recursive definitions for each E_i, where the recursive definition consists of sums of $E_j R_{ji}$ products, where R_{ji} is a regular expression consisting of the union of single characters. That is, R_{ji} represents the selection of single transitiosn from state j to state i, or single edges (j, i) in the graph. So if $\delta(q_j, a) = \delta(q_j, b) = q_i$, then $R_{ji} = (a + b)$. In other words, E_j takes the finite state automata from state q_0 to q_j. Then R_{ji} is a regular expression describing strings that will take the finite state automata from state j to state i in exactly one step. That is, intuitively:

$$E_i = \sum_{j \in Q} E_j R_{ji}.$$

Note: Recall that addition when dealing with regular expressions is the set union operation.

Once we have the system of equations, then we solve them by backwards substitution just as in linear algebra and high school algebra. We formalize our system of equations:

Definition 67 (Brzozowski Algebraic Method)**.** Let D be a DFA. Let $i, j \in Q$ and define $R_{ij} \subset (\Sigma \cup \{\epsilon\})$ where $R_{ij} = \{\omega : \delta(i, \omega) = j\}$. For each $i \in Q$, we define:

$$E_i = \sum_{q \in Q} E_q R_{qi}.$$

The desired regular expression is the closed form sum:

$$\sum_{f \in F} E_f.$$

In order to solve these equations, it is necessary to develop some machinery. More importantly, it is important to ensure this approach is sound. The immediate question to answer is whether E_q exists for each $q \in Q$? Intuitively, the answer is yes. Fix $q \in Q$. We take D, and construct a new DFA D_q whose state set, transitions, and initial state are the same as D. However, the sole final state of D_q is q. This is begging the question, though. We have shown that if a language is regular, then there exists a DFA that recognizes it.

The goal now is to show the converse: the language accepted by a finite state automaton is regular. To this end, we introduce the notion of a derivative for regular expressions, which captures the suffix relation. The derivative of a regular expression returns a regular expression, so it is accepted by some finite state automaton (which we have already proven in the forward direction of Kleene's Theorem). We view each R_{ij} as the sum of derivatives of each E_i. It then becomes necessary to show that a regular expression can be written as the sum of its derivatives. This implies the existence of a solution to the equations in Definition 67. With a high level view in mind, we proceed with the definition of a derivative.

Definition 68 (Derivative of Regular Expressions)**.** Let R be a regular expression and let $\omega \in \Sigma$. The derivative of R with respect to ω is $D_\omega R = \{t : \omega t \in L(R)\}$.

We next introduce the following function:

Definition 69. Let R be a regular expression, and define the function τ as follows:

$$\tau(R) = \begin{cases} \epsilon & : \epsilon \in L(R) \\ \emptyset & : \epsilon \not\in L(R). \end{cases} \tag{5}$$

Note that $\emptyset R = R\emptyset = \emptyset$, which follows from a counting argument. We note that $L(\emptyset R) = L(\emptyset) \times L(R) = \emptyset \times L(R)$. Now by the Rule of Product, $|\emptyset \times L(R)| = |\emptyset| \times |L(R)| = 0$.

It is clear the following hold for τ:

$$\begin{aligned}
&\tau(\epsilon) = \tau(R^*) = \epsilon \\
&\tau(a) = \emptyset \text{ for all } a \in \Sigma \\
&\tau(\emptyset) = \emptyset \\
&\tau(P + Q) = \tau(P) + \tau(Q) \\
&\tau(PQ) = \tau(P)\tau(Q).
\end{aligned}$$

Note that $\emptyset R = R\emptyset = \emptyset$, for any regular expression L. A simple counting argument for the cardinality of the concatenation of two languages justifies this.

Theorem 2.4. *Let $a \in \Sigma$ and let R be a regular expression. $D_a R$ is defined recursively as follows:*

$$D_a a = \epsilon \tag{6}$$
$$D_a b = \emptyset \textit{ if } b \in (\Sigma - \{a\}) \cup \{\epsilon\} \cup \{\emptyset\} \tag{7}$$
$$D_a(P^*) = (D_a P)P^* \tag{8}$$
$$D_a(PQ) = (D_a P)Q + \tau(P)D_a Q \tag{9}$$

Proof Sketch. Lines 6 and 7 follow immediately from Definition 68. We next show that the equation at line 8 is valid. Let $\sigma \in L(D_a(P^*))$. So we may write $\sigma = xy$, for some strings x, y. Without loss of generality, we assume that $ax \in L(P)$. So $x \in L(D_a P)$ and $y \in L(P^*)$. Thus, $L(\sigma \in (D_a P)P^*)$. Conversely, let $\psi \in L((D_a P)P^*)$. So $a\psi \in L(P^*)$, which implies that $D_a(a\psi) = \psi \in L(D_a P^*)$, as desired.

The proof of line 9 is left as an exercise for the reader. ☐

The next two results allow us to show that the derivative of a regular expression is itself a regular expression. So regular languages are preserved under the derivative operator. This algebraic proof is more succinct than modifying a FSM to accept a given language.

Theorem 2.5. *Let $\omega \in \Sigma^*$ with $|\omega| = n$, and let R be a regular expression. $D_\omega R$ satisfies:*

$$\begin{aligned}&D_\epsilon R = R\\&D_{\omega_1\omega_2} R = D_{\omega_2}(D_{\omega_1} R)\\&D_\omega R = D_{\omega_n}(D_{\omega_1\ldots\omega_{n-1}} R)\end{aligned}$$

Proof. This follows from the definition of the derivative and induction. We leave the details as an exercise for the reader. □

Theorem 2.6. *Let $s \in \Sigma^*$ and R be a regular expression. $D_s R$ is also a regular expression.*

Proof. The proof is by induction on $|s|$. When $|s| = 0$, $s = \epsilon$ and $D_s R = R$, which is a regular expression. Fix $k \geq 0$, and suppose that if $|s| = k$ then $D_s R$ is a regular expression. We prove true for the $k + 1$ case. Let ω be a string of length $k + 1$. From Theorem 2.5, $D_\omega R = D_{\omega_{k+1}}(D_{\omega_1\ldots\omega_k} R)$. By the inductive hypothesis, $(D_{\omega_1\ldots\omega_k} R)$ is a regular expression, which we call P. We apply the inductive hypothesis to $D_{\omega_{k+1}} P$ to obtain the desired result. □

We introduce the next theorem, which is easily proven using a set-inclusion argument. This theorem does not quite imply the correctness of the system of equations for the Brzozowski Algebraic Method. However, a similar argument shows the correctness of the Brzozowski system of equations.

Theorem 2.7. *Let R be a regular expression. Then:*

$$R = \tau(R) + \sum_{a \in \Sigma} a(D_a R).$$

In order to solve the Brzozowski system of equations, we use the substitution approach from high school. Because the set of regular languages forms a semi-ring, we do not have inverses for set union or string concatenation. So elimination is not a viable approach here. Arden's Lemma, which is given below, provides a means to obtain closed form solutions for each equation in the system. We then substitute the closed form solution into the remaining equations and repeat the procedure.

Lemma 2.4 (Arden)**.** *Let α, β be regular expressions and R be a regular expression satisfying $R = \alpha + \beta R$. Then $R = \alpha(\beta^*)$.*

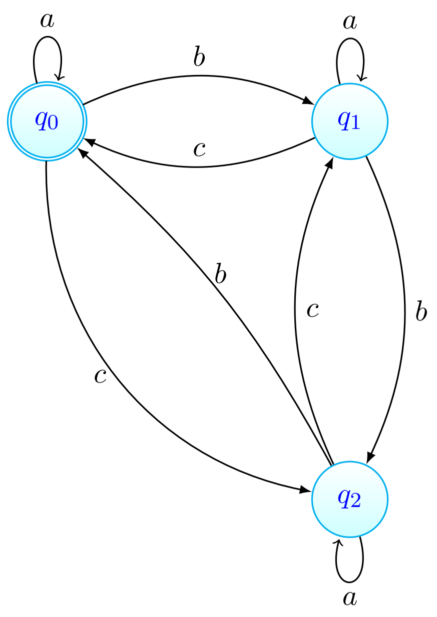
Remark: Arden's Lemma is analogous to homogenous first-order recurrence relations, which are of the form $a_0 = k$ and $a_n = ca_{n-1}$ where c, k are constants. The closed form solution for the recurrence is $a_n = kc^n$.

We now consider some examples of applying the Brzozowski Algebraic Method.

Example 64. We seek a regular expression over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ describing those integers whose value is 0 modulo 3.

In order to construct the finite state automata for this language, we take advantage of the fact that a number $n \equiv 0 \pmod 3$ if and only if the sum of n's digits are also divisible by 3. For example, we know $3|123$ because $1+2+3 = 6$, a multiple of 3. However, 125 is not divisible by 3 because $1+2+5 = 8$ is not a multiple of 3.

Now for simplicity, let's partition Σ into its equivalence classes $a = \{0, 3, 6, 9\}$ (values congruent to 0 mod 3), $b = \{1, 4, 7\}$ (values equivalent to 1 mod 3), and $c = \{2, 5, 8\}$ (values equivalent to 2 mod 3). Similarly, we let state q_0 represent a, state q_1 represent b, and state q_2 represent c. Thus, the finite state automata diagram is given below, with q_0 as the accepting halt state:



We consider the system of equations given by E_i, taking the FSM from state q_0 to q_i:

- $E_0 = \epsilon + E_0a + E_1c + E_2b$

  If at q_0, transition to q_0 if we read in the empty string, or if we go from $q_0 \to q_0$ and read in a character in a; or if we go from $q_0 \to q_2$ and read in a character in c; or if we go from $q_0 \to q_2$ and read in a character from b.

- $E_1 = E_0b + E_1a + E_2c$

  To transition from $q_0 \to q_1$, we can go from $q_0 \to q_0$ and read in a character from b; go from $q_0 \to q_1$ and read in a character from a; or go from $q_0 \to q_2$ and read in a character from c.

- $E_2 = E_0c + E_1b + E_2a$

  To transition from $q_0 \to q_2$, we can go from $q_0 \to q_0$ and read a character from c; go from $q_0 \to q_0$ and read in a character from b; or go from $q_0 \to q_2$ and read in a character from a.

Since q_0 is the accepting halt state, only a closed form expression of E_0 is needed.

There are two steps which are employed. The first is to simplify a single equation, then to backwards substitute into a different equation. We repeat this process until we have the desired closed-form solution for the relevant E_i (in this case, just E_0). In order to simplify a variable, we apply Arden's Lemma, which states that $E = \alpha + E\beta = \alpha(\beta)^*$, where α, β are regular expressions.

We start by simplifying E_2 using Arden's Lemma: $E_2 = (E_0c + E_1b)a^*$.

We then substitute E_2 into E_1, giving us $E_1 = E_0b + E_1a + (E_0c + E_1b)(a)^*c = E_0(b + ca^*c) + E_1(c + ba^*c)$. By Arden's Lemma, we get $E_1 = E_0(b + ca^*c)(a + ba^*c)^*$

Substituting again, $E_0 = \epsilon + E_0a + E_0(b + ca^*c)(a + ba^*c)^*c + (E_0c + E_1b)a^*b$.

Expanding out, we get $E_0 = \epsilon + E_0a + E_0(b + ca^*c)(a + ba^*c)^*c + E_0ca^*b + E_0(b + ca^*c)(a + ba^*c)^*a^*b$.

Then factoring out: $E_0 = \epsilon + E_0(a + ca^*b + (b + ca^*c)(a + ba^*c)^*(c + ba^*b))$.

By Arden's Lemma, we have: $E_0 = (a + ca^*b + (b + ca^*c)(a + ba^*c)^*(c + ba^*b))^*$, a closed form regular expression for the integers mod 0 over Σ.

Example 65. Consider the DFA:

State	Set	a	b
Q_0	$\{q_0\}$	Q_1	Q_2
$\star Q_1$	$\{q_0, q_2\}$	Q_1	Q_2
Q_2	$\{q_1\}$	Q_0	Q_3
$\star Q_3$	$\{q_1, q_2\}$	Q_0	Q_3

The Brzozowski Equations are shown below. We leave it as an exercise for the reader to solve this system of equations.

$$\begin{aligned}
E_0 &= E_2a + E_3a + \epsilon \\
E_1 &= E_0a + E_1a \\
E_2 &= E_0b + E_1b \\
E_3 &= E_2b + E_3b
\end{aligned}$$

2.7 Pumping Lemma for Regular Languages

So far, we have only examined languages which are regular. The Pumping Lemma for Regular Languages provides a non-deterministic test to determine if a language is not regular. We check if a language cannot be pumped. If this is the case, then it is not regular. However, there exist non-regular languages which satisfy the Pumping Lemma for Regular Languages. That is, the Pumping Lemma for Regular Languages is a necessary condition for a language to be regular. There are numerous Pumping Lemmas, including the Pumping Lemma for Context Free Languages and others in the literature for various classes of formal languages. So the Pumping Lemma for Regular Languages is a very natural result. We state it formally below.

Theorem 2.8 (Pumping Lemma for Regular Languages)**.** *Suppose L is a regular language. Then there exists a constant $p > 0$, depending only on L, such that for every string $\omega \in L$ with $|\omega| \geq p$, we can break w into three strings $w = xyz$ such that:*

- $|y| > 0$
- $|xy| \leq p$
- $xy^iz \in L$ *for all* $i \geq 0$

Proof. Let D be a DFA with p states accepting L, and let $\omega \in L$ such that $|\omega| \geq p$. We construct strings x, y, z satisfying the conclusions of the Pumping Lemma. Let $\hat{\delta}(\omega) = (q_0, \ldots, q_{|\omega|})$ be a complete accepting computation. As $|\omega| \geq p$, some state in the sequence $(q_0, \ldots, q_p)$ is repeated. Let $q_i = q_j$ with $i < j$ be repeated. Let $\omega_{i+1} \ldots \omega_j$ be the substring of ω taking the string from q_i to q_j. Let $x = \omega_1 \ldots \omega_i, y = \omega_{i+1} \ldots \omega_j$ and $z = \omega_{j+1} \ldots \omega_{|\omega|}$. So $|xy| \leq p$, $|y| > 0$ and $xy^kz \in L$ for every $k \geq 0$. □

We consider a couple examples to apply the Pumping Lemma for Regular Languages. In order to show a language is not regular, we pick one string and show that every decomposition of that string can be pumped to produce a new string not in the language. One strategy with pumping lemma proofs is to pick a sufficiently large string to minimize the number of cases to consider.

Proposition 2.1. *Let $L = \{0^n1^n : n \in \mathbb{N}\}$. L is not regular.*

Proof. Suppose to the contrary that L is regular and let p be the pumping length. Let $\omega = 0^p1^p$. By the Pumping Lemma for Regular Languages, there exist strings x, y, z such that $\omega = xyz$, $|xy| \leq p, |y| > 0$ and $xy^iz \in L$ for all $i \in \mathbb{N}$. Necessarily, $xy = 0^k$ for some $k \in [p]$, with y containing at least one 0. So $xy^0z = xz$ has fewer 0's than 1's. Thus, $xz \not\in L$, a contradiction. □

Proposition 2.2. *Let $L = \{0^n1^{2n}0^n : n \in \mathbb{N}\}$. L is not regular.*

Proof. Suppose to the contrary that L is regular and let p be the pumping length. Let $\omega = 0^p1^{2p}0^p$. By the Pumping Lemma for Regular Languages, let x, y, z be strings such that $\omega = xyz$, $|xy| \leq p$ and $|y| > 0$. Necessarily, xy contains only 0's and y contains at least one 0. So $xy^0z = xz$, which contains fewer than p 0's followed by $1^{2p}0^p$, so $xz \not\in L$, a contradiction. □

We examine one final example, which is a non-regular language that satisfies the Pumping Lemma for Regular Languages.

Example 66. Let L be the following language:

$$L = \{uvwxy : u, y \in \{0,1,2,3\}^*; v, w, x \in \{0,1,2,3\} \text{ s.t}(v = w \text{ or } v = x \text{ or } x = w)\} \cup$$
$$\{w : w \in \{0,1,2,3\}^* \text{ and precisely } 1/7 \text{ of the characters are } 3\}$$

Let p be the pumping length, and let $s \in L$ be a string with $|s| \geq p$. As the alphabet has order 4, there is a duplicate character within the first five characters. The first duplicate pair is separated by at most three characters. We consider the following cases:

- If the first duplicate pair is separated by at most one character, we pump one of the first five characters not separating the duplicate pair. As $u \in \{0,1,2,3\}^*$, the resultant string is still in L.
- If the duplicate pair is separated by two or three characters, we pump two consecutive characters separating them. If we pump down, we obtain a substring with the duplicate pair separated by either zero or one characters. If we pump the separators ab up, then we have aba in the new string. In both cases, the pumped strings belong to L.

However, L is not regular, which we show in the next section.

2.8 Closure Properties

The idea of closure properties is another standard idea in automata theory. So what exactly does closure mean? Informally, it means if we take two elements in a set and do something to them, we get an element in the set. This section focuses on operations on which regular languages are closed; however, we also have closure in other mathematical operations. Consider the integers, which are closed over addition. This means that if we take two integers and add them, we get an integer back.

Similarly, if we take two real numbers and multiply them, the product is also a real number. The real numbers are not closed under the square root operation, however. Consider $\sqrt{-1} = i$, which is a complex number but not a real number. This is an important point to note- operations on which a set is closed will never give us an element outside of the set. So adding two real numbers will never give us a complex number of the form $a + bi$ where $b \neq 0$.

Now let us look at operations on which regular languages are closed. Let Σ be an alphabet and let $RE(\Sigma)$ be the set of regular languages over Σ. A binary operator is closed on the set $RE(\Sigma)$ if it is defined as: $\odot : RE(\Sigma) \times RE(\Sigma) \to RE(\Sigma)$. In other words, each of these operations takes either one or two (depending on the operation) regular languages and returns a regular language. Note that the list of operations including set union, set intersection, set complementation, concatenation, and Kleene closure is by no means an extensive or complete list of closure properties.

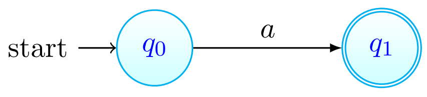
Recall from the definition of a regular language that if A and B are regular languages over the alphabet Σ, then $A \cup B$ is also regular. More formally, we can write $\cup : RE(\Sigma) \times RE(\Sigma) \to RE(\Sigma)$, which says that the set union operator takes two regular languages over a fixed alphabet Σ and returns a regular language over Σ. Similarly, string concatenation is a closed binary operator on $RE(\Sigma)$ where $A \cdot B = \{a \cdot b : a \in A, b \in B\}$. The set complementation and Kleene closure operations are closed, unary operators. Set complementation is defined as $- : RE(\Sigma) \to RE(\Sigma)$ where for a language $A \in RE(\Sigma)$, $\overline{A} = \Sigma^* \setminus A$. Similarly, the Kleene closure operator takes a regular language A and returns A^*.

Recall that the definition of a regular language provides for closure under the union, concatenation, and Kleene closure operations. The proof techniques rely on either modifying a regular expression or FSMs for the input languages. We have seen these techniques in the proof of Kleene's theorem. We begin with set complementation.

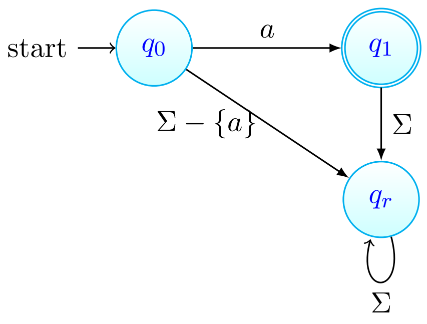
Proposition 2.3. *Let Σ be an alphabet. The set $RE(\Sigma)$ is closed under set complementation.*

Proof. Let L be a regular language, and let M be a DFA with a total transition function such that $L(M) = L$. We construct $\overline{M} = (Q_M, \Sigma, \delta_M, q_0, Q_M \setminus F_M)$. So $\omega \in \Sigma^*$ is not accepted by M if and only if ω is accepted by $\overline{M}$. So $\overline{L} = L(\overline{M})$, which implies that $\overline{L}$ is regular. □

Remark: It is important that the transition function is a total function; that is, it is fully defined. A partial transition function does not guarantee that this construction will accept $\overline{L}$. Consider the following FSM. The complement of this machine accepts precisely a.



However, if we fully define an equivalent machine (shown below), then the construction in the proof guarantees the complement machine accepts $\Sigma^* \setminus \{a\}$:



We now discuss the closure property of set intersection, for which two proofs are provided. The first proof leverages a product machine (similar to the construction of a product machine for the set union operation in the proof for Kleene's Theorem). The second proof uses existing closure properties, and so is much more succinct.

Proposition 2.4. *Let Σ be an alphabet. The set $RE(\Sigma)$ is closed under set intersection.*

Proof (Product Machine). Let L_1, L_2 be regular languages, and let M_1 and M_2 be fully defined DFAs that accept L_1 and L_2 respectively. We construct the product machine $M = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{01}, q_{02}), F_1 \times F_2)$. A simple set containment argument shows that $L(M) = L_1 \cap L_2$. We leave the details as an exercise for the reader. □

Proof (Closure Properties). Let L_1, L_2 be regular languages. By DeMorgan's Law, we have $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. As regular languages are closed under union and complementation, we have that $L_1 \cap L_2$ is regular. □

We next introduce the set difference closure property for two regular languages. Like the proof for the closure under set intersection, the proof of closure under set complementation relies on existing closure properties.

Proposition 2.5. *Let L_1, L_2 be regular languages. $L_1 \setminus L_2$ is also regular.*

Proof. Recall that $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$. As the set of regular languages is closed under intersection and complementation, $L_1 \setminus L_2$ is regular. □

We conclude with one final closure property before examining some examples of how to apply them.

Proposition 2.6. *Let L be a regular language, and let $L^R = \{\omega^R : \omega \in L\}$. We have L^R is regular.*

Proof. Let M be a DFA accepting L. We construct an ϵ-NFA M' to accept L^R as follows. The states of M' are $Q_M \cup \{q_0'\}$, where q_0' is the initial state of M'. We set $F' = \{q_0\}$, the initial state of M. Finally, for each transition $((q_i, s), q_j) \in \delta_M$, we add $((q_j, s), q_i) \in \delta_{M'}$. Finally, we add ϵ transitions from q_0' to each state $q_f \in F_M$. A simple set containment argument shows that $L(M') = L^R$, and we leave the details as an exercise for the reader. □

One application of closure properties is to show languages are or are not regular. To show a language fails to be regular, we operate on it with a regular language to obtain a known non-regular language. Consider the following example.

Example 67. Let $L = \{ww^R : w \in \{0,1\}^*\}$. Suppose to the contrary that L is regular. We Consider $L \cap 0^*1^*0^* = \{0^n1^{2n}0^n : n \in \mathbb{N}\}$. We know that $\{0^n1^{2n}0^n : n \in \mathbb{N}\}$ is not regular from Proposition 2.2 (recall from earlier this morning). As regular languages are closed under intersection, it follows that at least one of $0^*1^*0^*$ or L is not regular. Since $0^*1^*0^*$ is a regular expression, it follows that L is not regular.

The next example contains another non-regular language.

Example 68. Let $L = \{w :$ w contains an equal number of 0's and 1's $\}$. Consider $L \cap 0^*1^* = \{0^n1^n : n \in \mathbb{N}\}$. We know that $\{0^n1^n : n \in \mathbb{N}\}$ is not regular from Proposition 2.1. Since 0^*1^*, it follows that L is not regular.

We consider a third example.

Example 69. Let L be the language from Example 49 which satisfies the Pumping Lemma for Regular Languages.

$$L = \{uvwxy : u, y \in \{0,1,2,3\}^*; v, w, x \in \{0,1,2,3\} \text{ s.t}(v = w \text{ or } v = x \text{ or } x = w)\} \cup \tag{10}$$
$$\{w : w \in \{0,1,2,3\}^* \text{ and precisely } 1/7 \text{ of the characters are } 3\} \tag{11}$$

Consider:

$$L \cap (01(2+3))^* = \{w : w \in \{0,1,2,3\}^* \text{ and precisely } 1/7 \text{ of the characters are } 3\} \tag{12}$$

It is quite easy to apply the Pumping Lemma for Regular Languages to the language in (22), which implies that L is not regular.

We now use closure properties to show a language is regular.

Example 70. Let L be a regular language, and let $\overline{L}$ be its complement. Then $M = L \cdot \overline{L}$ is regular. As regular languages are closed under complementation, $\overline{L}$ is also regular. It follows that since regular languages are also closed under concatenation, that M is regular.

2.9 Myhill-Nerode and DFA Minimization

The Myhill-Nerode theorem is one of the most elegant and powerful results with respect to regular languages. It provides a characterization of regular languages, in addition to regular expressions and Kleene's theorem. Myhill-Nerode allows us to quickly test if a language is regular, and this test is deterministic. So we have a far more useful tool than the Pumping Lemma. Additionally, Myhill-Nerode implies an algorithm to minimize a DFA. The resultant DFA is not only *minimal*, in the sense that we cannot remove any states from it without affecting functionality; but it is also *minimum* as no DFA with fewer states that accepts the same language exists.

The intuition behind Myhill-Nerode is in the notion of distinguishing strings, with respect to a language as well as a finite state machine. We begin with the following definition.

Definition 70 (Distinguishable Strings)**.** Let L be a language over Σ. We say that two strings x, y are **distinguishable** w.r.t L if there exists a string z such that $xz \in L$ and $yz \notin L$ (or vice-versa).

Example 71. Let $L = \{0^n1^n : n \in \mathbb{N}\}$. The strings $0, 00$ are distinguishable with respect to L. Take $z = 1$. However, 0110 and 10 are not distinguishable, because $xz \notin L$ and $yz \notin L$ for every non-empty string z.

Example 72. Let $L = (0+1)^*1(0+1)$. The strings 00 and 01 are distinguishable with respect to L, taking $z = 0$.

We obtain a straight-forward result immediately from the definition of Distinguishable Strings.

Lemma 2.5. *Let L be a regular language, and let M be a DFA such that $L(M) = L$. Let x, y be distinguishable strings with respect to L. Then $M(x)$ and $M(y)$ end on different states.*

Proof. Suppose to the contrary that $M(x)$ and $M(y)$ terminate on the same state q. Let z be a string such that (WLOG) $xz \in L$ but $yz \notin L$. As D is deterministic, $M(xz)$ and $M(yz)$ transition from q_0 to q and then from q to some state q' on input z. So xz, yz are both in L, or xz, yz are both not in L. This contradicts the assumption that x, y are distinguishable. □

Remark: The above proof fails when using NFAs rather than DFAs, as an NFA may have multiple computations for a given input string.

We now introduce the notion of a distinguishable set of strings.

Definition 71 (Distinguishable Set of Strings)**.** A set of strings $\{x_1, \dots, x_k\}$ is distinguishable if every two distring strings x_i, x_j in the set are distinguishable.

This yields a lower bound on the number of states required to accept a regular language.

Lemma 2.6. *Suppose L is a language with a set of k distinguishable strings. Then every DFA accepting L must have at least k states.*

Proof. If L is not regular, no DFA exists and we are done. Let x_i, x_j be distinguishable strings. By Lemma 2.5, a DFA M run on x_i halts on a state q_i, while $M(x_j)$ halts on a different state q_j. So there are at least k states. □

We use Lemma 2.6 to show a language is in fact non-regular, by showing that for infinitely many $k \in \mathbb{N}$, there exists a set of k-distinguishable strings. Consider the following example.

Example 73. Recall that $L = \{0^n1^n : n \in \mathbb{N}\}$ is not regular. Let $\in \mathbb{N}$, and consider $S_k = \{0^i : i \in [k]\}$. Each of these strings is distinguishable. For $i \in [k]$, $z = 1^i$ distinguishes 0^i from the other strings in S_k. So for every $k \in \mathbb{N}$, we need a minimum of k states for a DFA to accept L. Thus, no DFA exists to accept L, and we conclude that L is not regular.

Definition 72. Let L be a language. Define $\equiv_L \subset \Sigma^* \times \Sigma^*$. Two strings x, y are said to be **indistinguishable** w.r.t. L, which we denote $x \equiv_L y$, if for every $z \in \Sigma^*$, $xz \in L$ if and only if $yz \in L$.

Remark: $\equiv_L$ is an equivalence relation. Note as well that $\Sigma^* / \equiv_L$ denotes the set of equivalence classes of $\equiv_L$. We now prove the Myhill-Nerode theorem.

Theorem 2.9 (Myhill-Nerode)**.** *Let L be a language over Σ. If Σ^* has infinitely many equivalence classes with respect to $\equiv_L$, then L is not regular. Otherwise, L is regular and is accepted by a DFA M where $|Q_M| = |\Sigma^* / \equiv_L|$.*

Proof. If Σ^* has an infinite number of equivalence classes with respect to $\equiv_L$, then we pick a string from each equivalence class. This set of strings is distinguishable. So by Lemma 2.6, no DFA exists to accept L. This shows that if L is regular, then $\Sigma^* / \equiv_L$ is finite.

Conversely, suppose $|\Sigma^* / \equiv_L|$ is finite. We construct a DFA M where Q is the set of equivalence classes of $\equiv_L$, Σ is the alphabet, $q_0 = [\epsilon]$, and $[\omega] \in F$ iff $\omega \in L$. We define the transition function $\delta([x], a) = [xa]$ for any $a \in \Sigma$. It suffices to show that δ is well-defined (that is, it is uniquely determined for any representative of an equivalence class). For any two strings $x \equiv_L y$, $[x] = [y]$ as $\equiv_L$ is an equivalence relation. We now show that $[xa] = [ya]$. Since $x \equiv_L y$, x, y are indistinguishable. So $xaz \in L$ iff $yaz \in L$ for every string z. So $[xa] = [ya]$. So the DFA is well-defined.

Finally, we show that $L(M) = L$. Let $x = x_1 \dots x_n$ be an input string. We run M on x. The resulting computation is $([\epsilon], [x_1], \dots, [x_1 \dots x_n])$. By construction of M, $x \in L$ if and only if $[x_1 \dots x_n] \in F$. So the DFA works as desired and L is regular. □

The Myhill-Nerode Theorem provides us with a *quotient machine* to accept L, though not a procedure to compute this machine explicitly. We show this DFA is minimum and unique, then discuss a minimization algorithm. We begin by defining a second equivalence relation.

Definition 73 (Distinguishable States)**.** Let M be a DFA. Define $\equiv_M$ to be a relation on $\Sigma^* \times \Sigma^*$ such that $x \equiv_M y$ if and only if $M(x)$ and $M(y)$ halt on the same state of M.

Remark: Observe that $\equiv_M$ is a second equivalence relation.

Theorem 2.10. *The machine M constructed by the Myhill-Nerode Theorem is minimum and unique up to relabeling.*

Proof. We begin by showing that M is minimal. Let D be another DFA accepting $L(M)$. Let $q \in Q(D)$. Let:

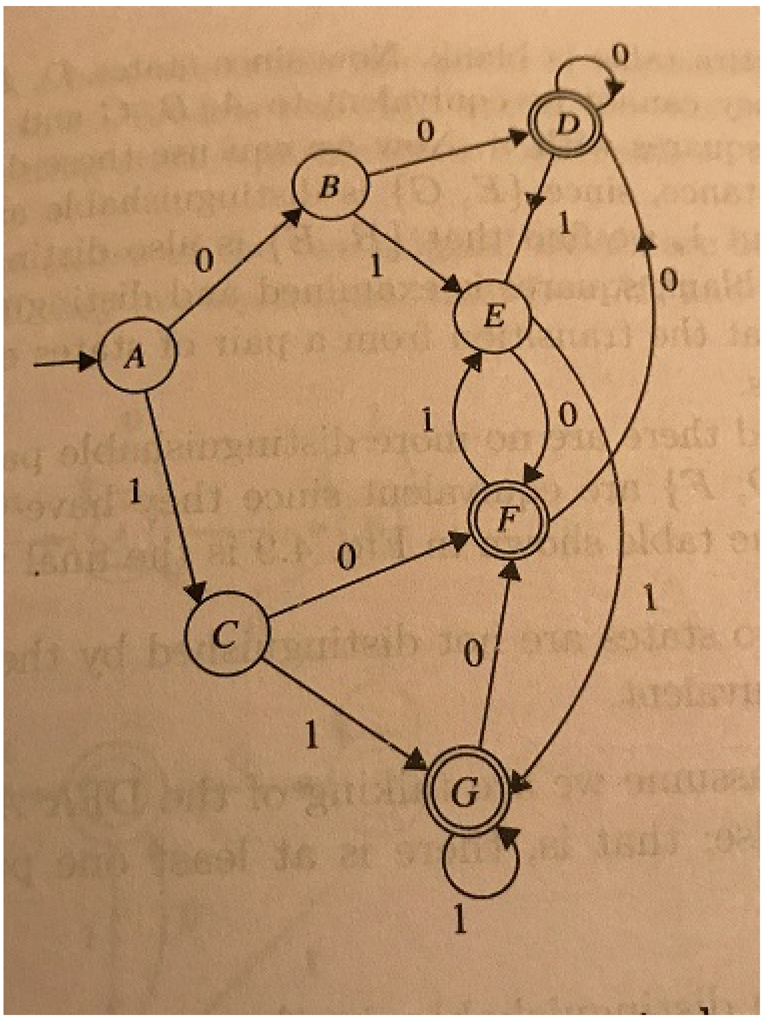
$$S_q = \{\omega : D(w) \text{ halts on } q\} \tag{13}$$

The strings in S_q are pairwise indistinguishable under $\equiv_L$. So $S_q \subset [x]$ for some $[x] \in Q(M)$. Thus, $|Q(D)| \geq |Q(M)|$.

We now show uniqueness. Suppose D has the same number of states as M but for some strings x, y, we have $x \equiv_D y$ but $x \not\equiv_M y$. Suppose M halts on state q when run on either x or y. Recall that $\equiv_M$ is the same relation as $\equiv_L$. So $[x]_M, [y]_M$ are distinct equivalence classes under $\equiv_M$. However, we have already shown that $S_q \subset [x]_M$, a contradiction. So M is the unique minimum DFA accepting L. $\square$

Computing $\equiv_L$ outright is challenging given the fact that Σ^* is infinite. We instead deal with $\equiv_M$ for a DFA M, which partitions Σ^* as well. Consider two states q_i, q_j of a M. Recall that S_{q_i} and S_{q_j} contain the set of strings such that M halts on q_i and q_j respectively when simulated on an input from the respective set. If the strings in S_{q_i} and S_{q_j} are indistinguishable, then S_{q_i} and S_{q_j} are subsets of the same equivalence class under $\equiv_L$. Thus, we can consolidate S_{q_i} and S_{q_j} into a single state. We again run into the same problem of determining which states of M are equivalent under $\equiv_L$. Instead, we deduce which states are not equivalent. This is done with a table-marking algorithm. We consider a $|Q| \times |Q|$ table, restricting attention to the bottom triangular half. The rows and columns correspond to states; and each cell is marked if and only if the two states are not equivalent. In each cell corresponding to a state in F and a state in $Q \setminus F$, we mark that cell. We then iterate until we mark no more states.

Example 74. We seek to minimize the following DFA:



We begin by noting that ϵ distinguishes the accept states and the non-accept states. So we mark the corresponding cells accordingly:

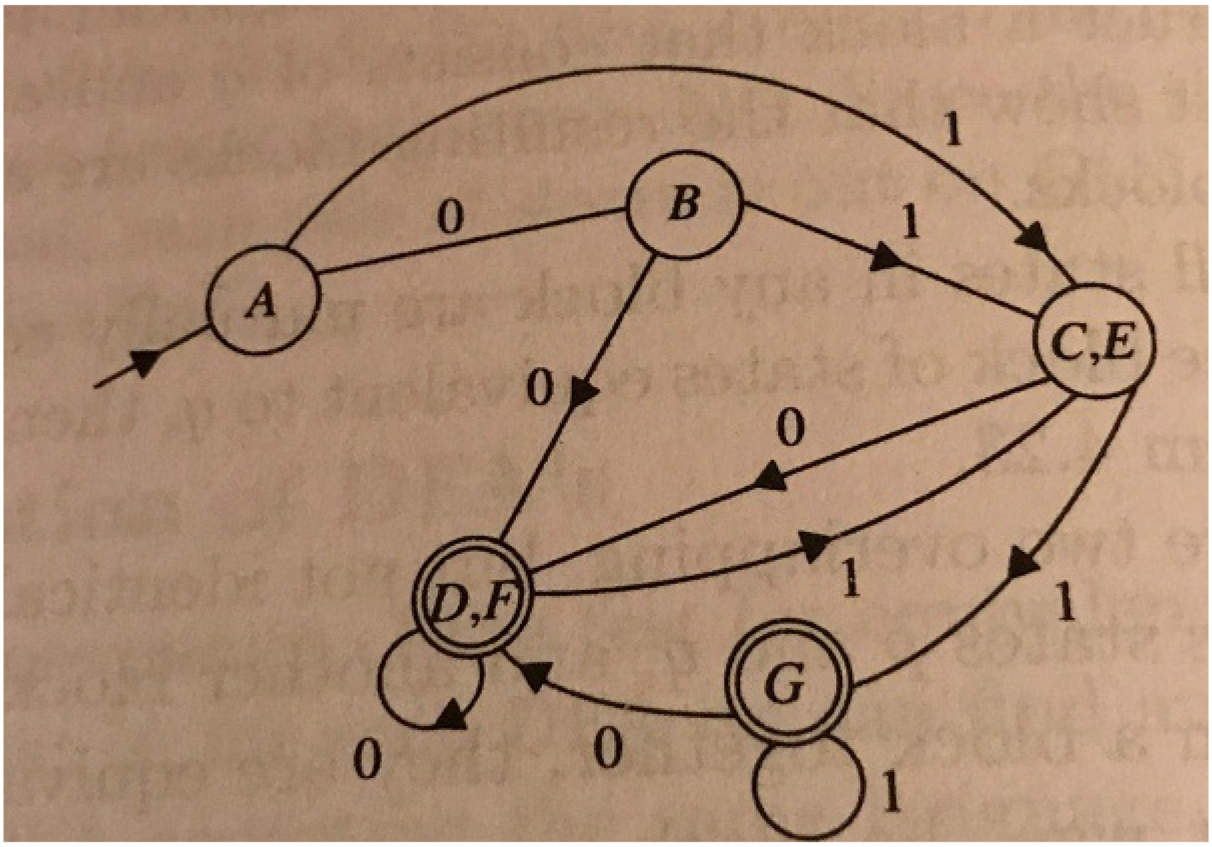
A						
	B					
		C				
X	X	X	D			
			X	E		
X	X	X		X	F	
X	X	X		X		G

We now deduce as many distinguishable states as possible:

- B and E are distinguishable states, as $\delta(B,1) = E$ and $\delta(E,1) = G$. As E and G are distinguishable states, it follows that B and E are distinguished by 1.
- F and G are distinguished by 1.
- D and G are distinguished by 1.
- A and B are distinguished by 0.
- A and C are distinguished by 0.
- A and E are distinguished by 0.
- B and C are distinguished by 1.
- As $\delta(C,x) = \delta(E,x)$ for $x \in \{0,1\}$, C and E are indistinguishable.
- As $\delta(D,x) = \delta(F,x)$ for $x \in \{0,1\}$, D and F are indistinguishable.

A						
X	B					
X	X	C				
X	X	X	D			
X	X		X	E		
X	X	X		X	F	
X	X	X	X	X	X	G

The minimal DFA:



3 More Group Theory (Optional)

In mathematics, we have various number systems with intricate structures. The goal of Abstract Algebra is to examine the common properties and generalize them into abstract mathematical structures, such as groups, rings, fields, modules, and categories. We restrict attention to groups, which are algebraic structures with an abstract operation of multiplication. Group theory has deep applications to algebraic combinatorics and complexity theory, with the study of group actions. Informally, a group action is a dynamical process on some set, which partitions the set into equivalence classes known as orbits. We study the structures of these orbits, which provide deep combinatorial insights such as symmetries of mathematical objects like graphs. This section is intended to supplement a year long theory of computation sequence in which elementary group theory is necessary, as well as provide a stand alone introduction to algebra for the eager mathematics or theoretical computer science student.

3.1 Introductory Group Theory

3.1.1 Introduction to Groups

In this section, we define a group and introduce some basic results. Recall that a group is an algebraic structure that abstracts over the operation of multiplication. Formally, we define a group as follows.

Definition 74. A group is an ordered pair $(G, \star)$ where $\star : G \times G \to G$ satisfies the following axioms:

- **Associativity:** For every $a, b, c \in G$, $(a \star b) \star c = a \star (b \star c)$
- **Identity:** There exists an element $1 \in G$ such that $1 \star a = a \star 1 = a$ for every $a \in G$.
- **Inverses:** For every $a \in G$, there exists an $a^{-1} \in G$ such that $a \star a^{-1} = a^{-1} \star a = 1$.

Definition 75 (Abelian Group)**.** A group $(G, \star)$ is said to be Abelian if $\star$ commutes; that is, if $a \star b = b \star a$ for all $a, b \in G$.

We have several examples of groups, which should be familiar. All of these groups are Abelian. However, we will introduce several important non-Abelian groups in the subsequent sections, including the Dihedral group, the Symmetry group, and the Quaternion group. In general, assuming a group is Abelian is both dangerous and erroneous.

Example 75. The following sets form groups using the operation of addition: $\mathbb{Z}, \mathbb{R}, \mathbb{Q}$, and $\mathbb{C}$.

Example 76. The following sets form groups using the operation of multiplication: $\mathbb{R} - \{0\}, \mathbb{Q} - \{0\}$, and $\mathbb{C} - \{0\}$. Note that $\mathbb{Z} - \{0\}$ fails to form a group under multiplication, as it fails to satisfy the inverses axiom. In particular, note that there does not exist an integer x such that $2x = 1$.

Example 77. Vector spaces form groups under the operation of addition.

Let's examine the group axioms more closely. Individually, each of these axioms seem very reasonable. Let's consider associativity at a minimum. Such a structure is known as a semi-group.

Definition 76 (Semi-Group)**.** A *semi-group* is a two-tuple $(G, \star)$ where $\star : G \times G \to G$ is associative.

Example 78. Let Σ be a finite set, which we call an alphabet. Denote Σ^* as the set of all finite strings formed from letters in Σ, including the empty string ϵ. Σ^* with the operation operation of string concatenation $\odot : \Sigma^* \times \Sigma^* \to \Sigma^*$ forms a semi-group.

Example 79. $\mathbb{Z}^+$ forms a semi-group under addition. Recall that $0 \not\in \mathbb{Z}^+$.

Associativity seems like a weak assumption, but it is quite important. Non-associative algebras are quite painful with which to work. Consider $\mathbb{R}^3$ with the cross-product operation. We show that this algebra does not contain an identity.

Proposition 3.1. *The algebra formed from $\mathbb{R}^3$ with the cross-product operation does not have an identity.*

Proof. Suppose to the contrary that there exists an identity $(x, y, z) \in \mathbb{R}^3$ for the cross-product operation. Let $(1, 1, 1)$ and (x, y, z) such that $(1, 1, 1) \times (x, y, z) = (1, 1, 1)$. Under the operation of the cross-product, we obtain:

$$\begin{aligned} z - y &= 1 \\ z - x &= 1 \\ x - y &= 1 \end{aligned}$$

So we obtain $z = y + 1$ and $z = x + 1$. However, $x = y + 1$ as well, so $x = z$, a contradiction. $\square$

In Example 76, we have already seen an algebra without inverses as well; namely, $\mathbb{Z} - \{0\}$ under the operation of multiplication. Imposing the identity axiom on top of the semi-group axioms gives us an algebraic structure known as a monoid.

Definition 77 (Monoid)**.** A *monoid* is an ordered pair $(G, \star)$ that forms a semi-group, and also satisfies the identity axiom of a group.

Example 80. Let S be a set, and let 2^S be the power set of S. The set union operation $\cup : 2^S \to 2^S$ forms a monoid.

Example 81. $\mathbb{N}$ forms a monoid under the operation of addition. Recall that $0 \in \mathbb{N}$.

Remark: Moving forward, we drop the $\star$ operator and simply write ab to denote $a \star b$. When the group is Abelian, we explicitly write $a + b$ to denote the group operation. This convention is from ring theory, in which we are dealing with two operations: addition (which forms an Abelian group) and multiplication (which forms a semi-group).

With some examples in mind, we develop some basic results about groups.

Proposition 3.2. *Let G be a group. Then:*

(A) The identity is unique.

(B) For each $a \in G$, a^{-1} is uniquely determined.

(C) $(a^{-1})^{-1} = a$ for all $a \in G$.

(D) $(ab)^{-1} = b^{-1}a^{-1}$

(E) For any $a_1, \ldots, a_n \in G$, $\prod_{i=1}^{n} a_i$ is independent of how the expression is parenthesized. (This is known as the generalized associative law).

Proof.

(A) Let $f, g \in G$ be identities. Then $fg = f$ since f is an identity, and $fg = g$ since g is an identity. So $f = g$ and the identity of G is unique.

(B) Fix $a \in G$ and let $x, y \in G$ such that $ax = ya = 1$. Then $y = y1 = y(ax)$. By associativity of the group operation, we have $y(ax) = (ya)x = 1x = x$. So $y = x = a^{-1}$, so a^{-1} is unique.

(C) Exchanging the role of a and a^{-1}, we have from the proof of (B) and the definition of an inverse that $a = (a^{-1})^{-1}$.

(D) We consider $abb^{-1}a^{-1} = a(bb^{-1})a^{-1}$ by associativity. By the group operation, $bb^{-1} = 1$, so $a(bb^{-1})a^{-1} = a(1)a^{-1} = aa^{-1} = 1$.

(E) The proof is by induction on n. When $n \leq 3$, the associativity axiom of the group yields the desired result. Now let $k \geq 3$; and suppose that for any $a_1, \ldots, a_k \in G$, $\prod_{i=1}^{k} a_i$ is uniquely determined, regardless of the parenthesization. Let $a_{k+1} \in G$ and consider $\prod_{i=1}^{k+1} a_i$. Any parenthesization of this product breaks it into two sub-products $\prod_{i=1}^{j} a_i$ and $\prod_{i=j+1}^{k+1} a_i$, each of which may be further parenthesized. As there are two non-empty products, each product has at most k terms. So by the inductive hypothesis, each subproduct is uniquely determined regardless of parenthesization. Let $x = \prod_{i=1}^{j} a_i$ and $y = \prod_{i=j+1}^{k+1} a_i$. We apply the inductive hypothesis again to xy to obtain the desired result.

□

We conclude this section with the definition of the *order* of a group and the definition of a subgroup.

Definition 78 (Order)**.** Let G be a group. We refer to $|G|$ as the *order of the group*. For any $g \in G$, we refer to $|g| = \min\{n \in \mathbb{Z}^+ : g^n = 1\}$ as the *order of the element* g. By convention, $|g| = \infty$ if no $n \in \mathbb{Z}^+$ satisfies $g^n = 1$.

Example 82. Let $G = \mathbb{Z}_6$ over addition. We have $|\mathbb{Z}_6| = 6$. The remainder class $\overline{3}$ has order 2 in G.

Example 83. Let $G = \mathbb{R}$ over addition. We have $|G| = \infty$ and $|g| = \infty$ for all $g \neq 0$. The order $|0| = 1$.

Definition 79 (Subgroup)**.** Let G be a group. We say that H is a subgroup of G if $H \subset G$ and H itself is a group. We denote the subgroup relation $H \leq G$.

Example 84. Let $G = \mathbb{Z}_6$ and $H = \{\overline{0}, \overline{3}\}$. So H is a subgroup of G, denoted $H \leq G$.

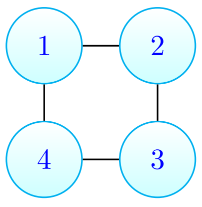
3.1.2 Dihedral Group

The Dihedral group is a non-Abelian group of key importance. In some ways, the Dihedral group is more tangible than the Symmetry group, and we shall see the reason for this shortly. Standard algebra texts introduce the Dihedral group as the group of symmetries for the regular polygon on n vertices, where $n \geq 3$. The Dihedral group provides an algebraic construction to study the rotations and reflections of the regular polygon on n vertices. This provides a very poor intuition for what constitutes a symmetry, or why rotations and reflections qualify here. Formally, a symmetry is a function from an object to itself that preservs one or more key underlying relations. More precisely, symmetries are automorphisms of a given structure. We begin with the definition of an isomorphism.

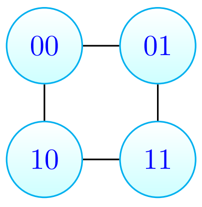
Definition 80 (Graph Isomorphism)**.** Let G, H be graphs. G and H are said to be *isomorphic*, denoted $G \cong H$, if there exists a bijection $\phi : V(G) \to V(H)$ such that $ij \in E(G) \implies \phi(i)\phi(j) \in E(H)$. The function ϕ is referred to as an *isomorphism*. The condition $ij \in E(G) \implies \phi(i)\phi(j) \in E(H)$ is referred to as the *homomorphism* condition. If $G = H$, then ϕ is a *graph automorphism*.

Remark: The Graph Isomorphism relation denotes that two graphs are, intuitively speaking, the same up to relabeling; this relation is an equivalence relation.

Example 85. We note that the graphs C_4 and Q_2 are isomorphic, denoted. $C_4 \cong Q_2$. Both C_4 and Q_2 are pictured below.

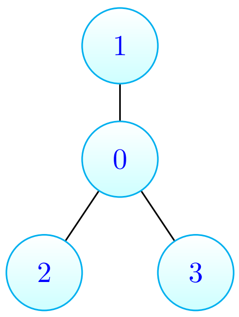


(a) Graph C_4



(b) Graph Q_2

Example 86. The graphs C_4 and $K_{1,3}$ are not isomorphic, denoted $C_4 \ncong K_{1,3}$. The graph C_4 is shown above in Example 85. We provide a drawing of $K_{1,3}$ below.



(a) Graph $K_{1,3}$

Remark: It is also important to note that graph homomorphisms, which are maps which for graphs G and H are maps $\phi : V(G) \to V(H)$ such that $ij \in E(G) \implies \phi(i)\phi(j) \in E(H)$, are of importance as well. One well-known class of graph homomorphisms are known as *colorings*, which are labelings of the vertices such that no two adjacent vertices use the same color. Suppose in particular we color a graph G using ℓ colors. We may view these ℓ colors as the vertices of K_ℓ. So an ℓ-coloring of G is simply a graph homomorphism $\varphi : V(G) \to V(K_\ell)$.

The *chromatic number* of a graph G, dentoed $\chi(G)$, is the minimum number of colors t such that there exists a graph homomorphism $\varphi : V(G) \to V(K_t)$. Determining the chromatic number of a graph is an `NP`-Hard problem. We will discuss graph homomorphisms in greater detail at a later point, as well as group isomorphisms and group homomorphisms. For now, we proceed to discuss the Dihedral group.

Recall that the Dihedral group is the group of rotations and reflections of the regular polygon on n vertices (where $n \geq 3$). This geometric object is actually the cycle graph C_n, and the rotations and reflections are actually automorphisms. We start by intuiting the structure of the automorphisms for cycle graphs. As an automorphism preserves a vertex's neighbors, it is necessary that a vertex v_i can only map to a vertex with degree at least $\deg(v_i)$. As the cycle graph is 2-regular (all vertices have degree 2), there are no restrictions in terms of mapping a vertex v_i of higher degree to a vertex v_j of lower degree (as this would result in one of v_i's adjacencies not being preserved). We next look more closely at rotations and reflections.

- **Rotations.** A *rotation* of the cycle graph C_n is a function $r : V(C_n) \to V(C_n)$ such that for a vertex $v_i \in V(C_n)$, $r(v_i) = v_{i+1}$, where the indices are taken modulo n. That is, a single rotation of C_n sends $v_1 \mapsto v_2$, $v_2 \mapsto v_3$, $\ldots$, $v_n \mapsto v_1$. Notice that the rotation map r is a bijection. Furthermore, observe that the rotation map preserves the neighbors of a given vertex.

  Now consider the composition $r \circ r$, which we denote r^2. Here, $r^2(v_1) = v_3$, $r^2(v_2) = v_4$, $\ldots$, $r^2(v_n) = v_2$. In other words, r^2 simply applies the rotation operator twice. More generally, for any $i \in \{0, 1, \ldots, n-1\}$, $r^i(v_1) = v_i$, $r^i(v_2) = v_i$, and so on. Here, we view r^0 as the identity map, which we denote 1 (that is, the identity map will be the identity of the Dihedral group). In particular, we observe that it requires exactly n rotations of the n cycle in order to send $v_i \mapsto v_i$ for all $i \in [n]$. So there are n distinct rotations of the n-cycle: $\{1, r, r^2, r^3, \ldots, r^{n-1}\}$. We can denote this set more concisely by specify the generator and its order, as follows:
$$\langle r : r^n = 1 \rangle.$$

  Here, r is the generator, and $r^n = 1$ indicates that $|r| = 1$. The generator r can be multiplied by itself (i.e., composed with itself) arbitrarily (but only finitely) many times. We reduce these products using the relation $r^n = 1$ and keep only the distinct elements. So for example, $r^{3n+2} = r^2$ and $r^{5n-1} = r^{n-1}$. As a result, we have that:
$$\langle r : r^n = 1 \rangle = \{1, r, r^2, r^3, \ldots, r^{n-1}\}.$$

  In particular, $\langle r : r^n = 1 \rangle$ is a subgroup of $\text{Aut}(C_n)$. We also note that $r^{-1} = r^{n-1}$. We may think of r^{-1} as undoing a rotation to the right. This moves vertex v_i back from position v_{i+1} to its original position. We may also acheive this same result simply by rotating the vertices to the right an additional $n-1$ units. So r^{n-1} acheives the same result as r^{-1}.

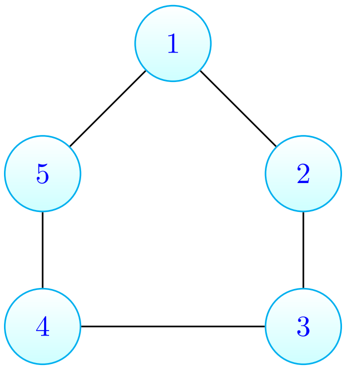
- **Reflections:** In C_n, v_1 has neighbors v_2, v_n. So in the automorphism, $\phi(v_n)\phi(v_1), \phi(v_1)\phi(v_2) \in E(C_n)$. This leaves two options. Either preserve the sequence: $\phi(v_n) - \phi(v_1) - \phi(v_2)$ or swap the vertices v_n, v_2 under automorphism to get the sequence $\phi(v_2) - \phi(v_1) - \phi(v_n)$. This gives rise to the *reflection.* As an automorphism is an isomorphism, it must preserve adjacencies. So the sequence $\phi(v_2) - \phi(v_3) - \ldots - \phi(v_{n-1}) - \phi(v_n)$ must exist after mapping v_1. After fixing v_1, v_2 and v_n, there is only one option for each of $\phi(v_3)$ and $\phi(v_{n-1})$. This in terms fixes $\phi(v_4)$ and $\phi(v_{n-2})$, etc. In short, fixing a single vertex then choosing whether or not to reflect its adjacencies fixes the entire cycle under automorphism.

  Conceptually, consider taking n pieces of rope and tying them together. The knots are analogous to the graph vertices. If each person holds a knot, shifting the people down by n positions is still an isomorphism. A single person can then grab the two incident pieces of rope to his or her knot, and flip those pieces around (the reflection operation). The same cycle structure on the rope remains. This is in fact, the fundamental idea of reflection. Furthermore, we observe that a second reflection undoes the first one. So the reflection operator, which we denote s, has order 2. So the group of reflections is

$$\{1, s\} = \langle s : s^2 = 1 \rangle.$$

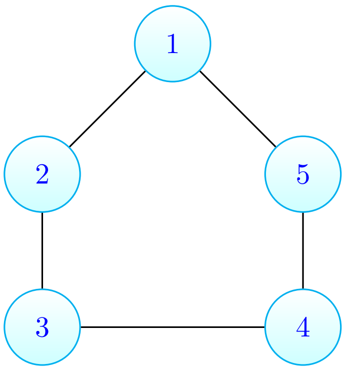
We now introduce the Dihedral group. Certainly, the Dihedral group is generated by r and s, with the relations established above, that $r^n = s^2 = 1$. We need a third relation to describe how r and s interact; namely, $rs = sr^{-1}$. Recall that rs is really the composition $r \circ s$. In other words, in rs, we reflect first and then rotate the vertices of C_n in the forward direction. Similarly, sr^{-1} first rotates the vertices of C_n in the backwards direction prior to performing the reflection. We provide an example of rs and sr^{-1} acting on C_5 to illustrate the concept.

Example 87. Consider the graph C_5.

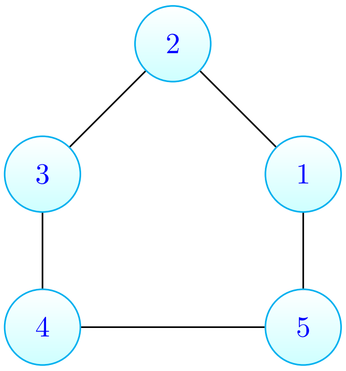


(a) Graph C_5

We next apply rs to C_5.

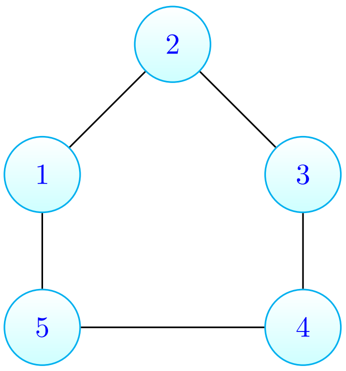


(b) C_5 after applying s.

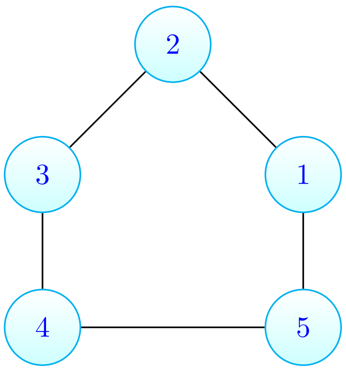


(c) C_5 after applying rs

We next apply sr^{-1} to the original C_5 in Figure (a).



(d) C_5 after applying r^{-1}.



(e) C_5 after applying sr^{-1}

Definition 81 (Dihedral Group)**.** Let $n \geq 3$. The *Dihedral group of order* $2n$, denoted D_{2n}, is given by the following presentation:

$$D_{2n} = \langle r, s : r^n = s^2 = 1, rs = sr^{-1} \rangle.$$

Remark: Formally, $D_{2n} \cong \text{Aut}(C_n)$, where $\text{Aut}(C_n)$ is the automorphism group of the cycle graph C_n. Precisely, we have only shown that D_{2n} is contained in $\text{Aut}(C_n)$. That is, we have shown that the rotation group $\langle r : r^n = 1 \rangle$ and the reflection group $\langle s : s^2 = 1 \rangle$ are all automorphisms of C_n. As $r, s \in \text{Aut}(C_n)$, it follows that the group generated by r and s, namely D_{2n}, is contained in $\text{Aut}(C_n)$. To show that $D_{2n} \cong \text{Aut}(C_n)$, we need a tool known as the Orbit-Stabilizer Theorem, which will be discussed later.

The presentation of the Dihedral group captures the notions of rotation and reflection which have been discussed so far. The final relation in the presentation of D_{2n}, $rs = sr^{-1}$, provides sufficient information to compute the inverse of each element. The presentation states that $(rs)^2 = 1$, which implies that $rs = (rs)^{-1}$. Let's solve for the exact form of $(rs)^{-1}$. By the presentation of $D_{2n}, s^2 = 1$, which implies that $|s| \leq 2$. As $s \neq 1, |s| = 2$. So $rs \cdot s = r$. As $r^n = 1$, it follows that $r^{n-1} = r^{-1}$. Intuitively, r states to rotate the cycle by one element clockwise. So r^{-1} undoes this operation. Rotating one unit counter-clockwise will leave the cycle in the same state as rotating it $n-1$ units clockwise. So $r^{-1} = r^{n-1}$. It follows that $rs = (rs)^{-1} = sr^{-1} = sr^{n-1}$. And so $(r^i s)^{-1} = sr^{-i} = sr^{n-i}$.

We conclude with a final remark about what is to come. The Dihedral group provides our first example of a group action. Intuitively, a group action is a dynamical process in which a group's elements are used to permute the elements of some set. We study the permutation structures which arise from the action, which provide deep combinatorial insights. Here, the Dihedral group acts on the cycle graph C_n by rotating and reflecting its vertices. This is a very tangible example of the dynamical process of a group action. We will formally introduce group actions later.

3.1.3 Symmetry Group

The Symmetry group is perhaps one of the most important groups, from the perspective of algebraic combinatorics. The Symmetry group captures all possible permutations on a given set. Certain subgroups of the Symmetry group provide extensive combinatorial information about the symmetries of other mathematical objects. We begin by formalizing the notion of a permutation.

Definition 82 (Permutation)**.** Let X be a set. A permutation is a bijection $\pi : X \to X$.

Formally, we define the Symmetry group as follows:

Definition 83 (Symmetry Group)**.** Let X be a set. The *Symmetry group* $\text{Sym}(X)$ is the set of all permutations $\pi : X \to X$ with the operation of function composition.

Remark: Recall that the composition of two bijections is itself a bijection. This provides for closure of the group operation. The identity map is a bijection, and so a permutation. This is the identity of the Symmetry group. In order to see that the Symmetry group is closed under inverses, it is helpful to think of a permutation as a series of swaps or transpositions. We simply undo each transposition to obtain the identity. Showing that every permutation can be written as the product of (not necessarily disjoint) two-cycles is an exercise left to the reader.

We first show how to write permutations in cycle notation. Formally, we have the following definition.

Definition 84 (Cycle)**.** A *cycle* is a sequence of distinct elements which are cyclically permuted.

Definition 85 (Cycle Decomposition)**.** The *cycle decomposition* of a permutation π is a sequence of cycles where no two cycles contain the same elements. We refer to the cycle decomposition as a product of disjoint cycles, where each cycle is viewed as a permutation.

Theorem 3.1. *Every permutation π of a finite set X can be written as the product of disjoint cycles.*

The proof is constructive. We provide an algorithm to accomplish this. Intuitively, a cyclic permutation is simply a rotation. So we take an element $x \in X$ and see where x maps under π. We then repeat this for $\pi(x)$. The cycle is closed when $\pi^n(x) = x$. Formally, we take $(x, \pi(x), \pi^2(x), \ldots, \pi^n(x))$. We then remove the elements covered by this cycle and repeat for some remaining element in X. As X is finite and each iteration partitions at least one element into a cycle, the algorithm eventually terminates. The correctness follows from the fact that this construction provides a bijection between permutations and cycle decompositions. We leave the details of this to the reader, but it should be intuitively apparent. Let's consider a couple examples.

Example 88. Let σ be the permutation:

$$1 \mapsto 3 \quad 2 \mapsto 4 \quad 3 \mapsto 5 \quad 4 \mapsto 2 \quad 5 \mapsto 1$$

Select 1. Under σ, we have $1 \mapsto \sigma(1) = 3$. Then $3 \mapsto \sigma(3) = 5$. Finally, $5 \mapsto \sigma(5) = 1$. So we have one cycle $(1, 3, 5)$. By similar analysis, we have $2 \mapsto 4$ and $4 \mapsto 2$, so the other cycle is $(2, 4)$. Thus, $\sigma = (1, 3, 5)(2, 4)$.

Example 89. Let τ be the permutation:

$$1 \mapsto 5 \quad 2 \mapsto 3 \quad 3 \mapsto 2 \quad 4 \mapsto 4 \quad 5 \mapsto 1 \tag{14}$$

Select 1. We have $1 \mapsto 5$, and then $5 \mapsto 1$. So we have the cycle $(1, 5)$. Similarly, we have the cycle $(2, 3)$. Since $4 \mapsto 4$, we have (4). By convention, we do not include cycles of length 1 which are fixed points. So $\tau = (1, 5)(2, 3)$.

In order to deal with the cycle representation in any meaningful way, we need a way to evaluate the composition of two permutations, which we call the **Cycle Decomposition Procedure**. We provide a second algorithm to take the product of two cycle decompositions and produce the product of disjoint cycles. Consider two permutations σ, τ and evaluate their product $\sigma\tau$, which is parsed from right to left as the operation is function composition. We view each cycle as a permutation and apply a similar procedure as above. We select an element x not covered in the final answer and follow it from right-to-left according to each cycle. When we reach the left most cycle, the element we end at is x's image under the product permutation. We then take x's image and repeat the procedure until we complete the cycle; that is, until we end back at x. We iterate again on some uncovered element until all elements belong to disjoint cycles. We consider an example.

Example 90. We consider the permutation

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

We evaluate this permutation as follows.

- We begin by selecting 1 and opening a cycle (1. Under $(1, 2, 3, 4, 5)$ we see $1 \mapsto 2$. We then move to $(1, 3, 4)$, under which 2 is a fixed point. Then under $(2, 4)$, $2 \mapsto 4$. Next, we see $4 \mapsto 5$ under $(3, 4, 5)$. Then $5 \mapsto 3$ under $(3, 5)$, and $3 \mapsto 1$ under $(1, 2, 3)$. So 1 is a fixed point and we close the cycle: (1).
- Next, we select 2. Under $(1, 2, 3, 4, 5)$, $2 \mapsto 3$. We then see $3 \mapsto 4$ under $(1, 3, 4)$. Under $(2, 4), 4 \mapsto 2$. The cycle $(3, 4, 5)$ fixes 2, as does $(3, 5)$. So we finally have $2 \mapsto 3$, yielding $(2, 3$.
- By similar analysis, we see $3 \mapsto 4 \mapsto 1 \mapsto 2$, so we close $(2, 3)$.
- The only two uncovered elements are now 4 and 5. Selecting 4, we obtain $4 \mapsto 5 \mapsto 3 \mapsto 5$. So we have $(4, 5$. Then we see $5 \mapsto 1 \mapsto 3 \mapsto 4$, so we close $(4, 5)$.

Thus:

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

The Cycle Decomposition Algorithm provides us with a couple nice facts.

Theorem 3.2. *Let c_1, c_2 be disjoint cycles. Then $c_1c_2 = c_2c_1$.*

Proof. We apply the Cycle Decomposition algorithm to c_1c_2 starting with the elements in c_2, to obtain c_2c_1. □

Remark: This generalizes for any product of n disjoint cycles.

Theorem 3.3. *Let $(x_1, \ldots, x_n)$ be a cycle. Then $|(x_1, \ldots, x_n)| = n$.*

Proof. Consider $(x_1, \ldots, x_n)^n = \prod_{i=1}^{n}(x_1, \ldots, x_n)$. Applying the cycle decomposition algorithm, we see $x_1 \mapsto x_2 \mapsto \ldots \mapsto x_n \mapsto x_1$. We iterate on this procedure for each element in the cycle to obtain $\prod_{i=1}^{n}(x_1, \ldots, x_n) = (1)$, the identity permutation. So $|(x_1, \ldots, x_n)| \leq n$. Applying the cycle decomposition procedure to $(x_1, \ldots, x_n)^k$ for any $k \in [n-1]$, and we see $x_1 \mapsto x_k, x_2 \mapsto x_{k+1}, \ldots, x_n \mapsto x_{n+k+1}$ where the indices are taken modulo n. □

3.1.4 Group Homomorphisms and Isomorphisms

In the exposition of the Dihedral group, we have already defined the notion of a graph isomorphism. Recall that a graph isomorphism is a bijection from the vertex sets of two graphs G and H, that preserves adjacencies. The group isomorphism is defined similarly. The one change is the notion of a homomorphism. While a graph homomorphism preserves adjacencies, a group homomorphism preserves the group operation. Formally:

Definition 86 (Group Homomorphism)**.** Let $(G, \star)$ and $(H, \diamond)$ be groups. A *group homomorphism* from G to H is a function $\phi : G \to H$ such that $\phi(x \star y) = \phi(x) \diamond \phi(y)$ for all $x, y \in G$. Omitting the formal operation symbols (as is convention), the homomorphism condition can be written as $\phi(xy) = \phi(x)\phi(y)$.

Example 91. Let $G = \mathbb{Z}_4$ and $H = \mathbb{Z}_2$. The function $\phi : \mathbb{Z}_4 \to \mathbb{Z}_2$ sending $\overline{0}_4$ and $\overline{2}_4$ to $\overline{0}_2$, and $\overline{1}_4$ and $\overline{3}_4$ to $\overline{1}_2$ is a homomorphism. We verify as follows. Let $\overline{x}_4, \overline{y}_4 \in \mathbb{Z}_4$. We have $\overline{x}_4 + \overline{y}_4 = \overline{x+y}_4$. We have $\overline{x+y}_2 = \overline{0}_2$ if and only if $\overline{x+y}_4 \in \{\overline{0}_4, \overline{2}_4\}$ if and only if $\overline{x}_4 + \overline{y}_4 \in \{\overline{0}_4, \overline{2}_4\}$. So ϕ is a homomorphism.

We now introduce the notion of a group isomorphism.

Definition 87 (Group Isomorphism)**.** Let G, H be groups. A *group isomorphism* is a bijection $\phi : G \to H$ that is also a group homomorphism. We say that $G \cong H$ (G is isomorphic to H) if there exists an isomorphism $\phi : G \to H$.

We consider a couple examples of group isomorphisms.

Example 92. Let G be a group. The identity map $\text{id} : G \to G$ is an isomorphim.

Example 93. Let $\exp : \mathbb{R} \to \mathbb{R}^+$ be given by $x \mapsto e^x$. This map is a bijection, as we have a well-defined inverse function: the natural logarithm. We easily verify the homomorphism condition: $\exp(x+y) = e^{x+y} = e^x e^y$.

There are several important problems relating to group isomorphisms:

- Are two groups isomorphic?
- How many isomorphisms exist between two groups?
- Classify all groups of a given order.

In some cases, it is easy to decide if two groups are (not) isomorphic. In general, the group isomorphism problem is undecidable; no algorithm exists to decide if two arbitrary groups are isomorphic. In particular, if two groups $G \cong H$, the following necessary conditions hold:

- $|G| = |H|$
- G is Abelian if and only if H is Abelian
- $|x| = |\phi(x)|$ for every $x \in G$ and every isomorphism $\phi : G \to H$.

It is quite easy to verify that the isomorphism relation preserves commutativity. Consider an isomorphism $\phi : G \to H$ and let $a, b \in G$. If G is Abelian, then $ab = ba$. Applying the isomorphism, we have $\phi(a)\phi(b) = \phi(b)\phi(a)$. As ϕ is surjective, it follows that H is also Abelian.

Similarly, it is quite easy to verify that for any isomorphism ϕ that $|x| = |\phi(x)|$. We prove a couple lemmas.

Lemma 3.1. *Let G, H be groups and let $\phi : G \to H$ be a homomorphism. Then $\phi(1_G) = 1_H$.*

Proof. Recall that $\phi(1_G) = \phi(1_G \cdot 1_G)$. Applying the homomorphism, we obtain that $\phi(1_G) = \phi(1_G)\phi(1_G) = \phi(1_G) \cdot 1_H$. By cancellation, we obtain that $\phi(1_G) = 1_H$. □

With Lemma 3.1 in mind, we prove this next Lemma.

Lemma 3.2. *Let G, H be groups and let $\phi : G \to H$ be a homomorphism. Then $|x| \geq |\phi(x)|$.*

Proof. Let $n = |x|$. So $\phi(x^n) = \phi(x)^n = \phi(1_G) = 1_H$. Thus, $|\phi(x)| \leq n$. □

We now show that isomorphisms are closed under inverses.

Theorem 3.4. *Let G, H be groups and let $\phi : G \to H$ be an isomorphism. Then $\phi^{-1} : H \to G$ is also an isomorphism.*

Proof. An isomorphism is a bijection, so $\phi^{-1} : H \to G$ exists and is a function. It suffices to show that ϕ^{-1} is a homomorphism. Let $a, b \in G$ and $c, d \in H$ such that $\phi(a) = c$ and $\phi(b) = d$. So $\phi(ab) = \phi(a)\phi(b) = cd$. We apply ϕ^{-1} to obtain $\phi^{-1}(cd) = \phi^{-1}(\phi(a)\phi(b)) = \phi^{-1}(\phi(ab))$, with the last equality as ϕ is a homomorphism. So $\phi^{-1}(\phi(ab)) = ab$. Similarly, $\phi^{-1}(c)\phi^{-1}(d) = ab$. So ϕ^{-1} is a homomorphism, and therefore an isomorphism. □

We now use Lemma 3.2 and Theorem 3.4 to deduce that isomorphism preserves each element's order.

Theorem 3.5. *Let G, H be groups and let $\phi : G \to H$ be an isomorphism. Then $|x| = |\phi(x)|$ for all $x \in G$.*

Proof. We have $|x| \geq |\phi(x)|$ from Lemma 3.2. By Theorem 3.4, ϕ^{-1} is an isomorphism. So we interchange the roles of x and $\phi(x)$ and apply Lemma 3.2, to obtain that $|\phi(x)| \geq |x|$. □

We conclude this section with a classification result. The proof of this theorem requires machinery we do not presently have; namely, Lagrange's Theorem which states that for any subgroup H of a finite group G, $|H|$ divides $|G|$. We defer the proof of Lagrange's Theorem until the next section.

Theorem 3.6. *Every group of order 6 is either S_3 or $\mathbb{Z}_6$.*

3.1.5 Group Actions

The notion of a group action is one of the most powerful and useful notions from algebra. Intuitively, a group action is a discrete dynamical process on a set of elements that partitions the set. The structure and number of these equivalence classes provide important insights in algebra, combinatorics, and graph theory. We formalize the notion of a group action as follows.

Definition 88 (Group Action)**.** Let G be a group and let A be a set. A *group action* is a function $\cdot : G \times A \to A$ (written $g \cdot a$ for all $g \in G$ and $a \in A$) satisfying:

1. $g_1 \cdot (g_2 \cdot a) = (g_1 g_2) \cdot a$ for all $g_1, g_2 \in G$ and all $a \in A$.
2. $1 \cdot a = a$ for all $a \in A$

We first consider several important examples of group actions:

Example 94. Let G be a group and let A be a set. The action in which $g \cdot a = a$ for all $g \in G$ and all $a \in A$ is known as the *trivial action*. In this case, $\sigma_g = (1)$ for all $g \in G$. The trivial action provides an example of why it is sufficient for G to act on itself in order to establish an isomorphism to a permutation group.

Example 95. Let A be a set and let $G = \text{Sym}(A)$. The action of $\text{Sym}(A)$ on A is given by $\sigma \cdot a = \sigma(a)$ for any permutation σ and element $a \in A$.

Example 96. Let $G = D_{2n}$ and let $A = V(C_n)$, the vertex set of the cycle graph C_n. D_{2n} acts on the vertices of C_n by rotation and reflection. In particular, $r = (1, 2, \ldots, n)$ and $s = \prod_{i=2}^{\lfloor n/2 \rfloor}(i, n - i + 1)$.

Before providing examples of group actions, we begin by proving Cayley's Theorem which yields that every group action has a permutation representation. That is, if the group G acts on the set A, G permutes A. Formally, we have the following.

Theorem 3.7 (Cayley's Theorem)**.** *Let G act on the set A. Then there exists a homomorphism from G into Sym(A). When $A = G$ (that is, when G is acting on itself), we have an isomorphism from G to a group of permutations.*

Proof. For each $g \in G$, we define the map $\sigma_g : A \to A$ by $\sigma_g(a) = g \cdot a$. We prove the following propositions.

Proposition 3.3. *For each $g \in G$, the function σ_g is a permutation. Furthremore, $\{\sigma_g : g \in G\}$ forms a group.*

Proof. In order to show that σ_g is a permutation, it suffices to show that σ_g has a two-sided inverse. Consider $\sigma_{g^{-1}}$, which exists as G is a group. We have $(\sigma_{g^{-1}} \circ \sigma_g)(a) = g^{-1} \cdot (g \cdot a) = (g^{-1}g) \cdot a = a$. So $\sigma_{g^{-1}} \circ \sigma_g = (1)$, the identity map. As g was arbitrary, we exchange g and g^{-1} to obtain that $\sigma_g \circ \sigma_{g^{-1}} = (1)$ as well. So σ_g has a two-sided inverse, and so it is a permutation. As G is a group, we have that $H = \{\sigma_g : g \in G\}$ is non-empty and closed under inverses, with $\sigma_1 \in H$ as the identity. As each $\sigma_i, \sigma_j \in H$ is a permutation of A, $\sigma_i \circ \sigma_j$ is also a permutation of A. In particular, for any $i, j \in G$ and any $a \in A$, we have that:

$$\begin{aligned}
&(\sigma_i \circ \sigma_j)(a) \\
&= \sigma_i(j \cdot a) \\
&= i \cdot (j \cdot a) \\
&= ij \cdot a \\
&= \sigma_{ij}(a).
\end{aligned}$$

So $\sigma_i \circ \sigma_j = \sigma_{ij}$. As G is a group, $ij \in G$. So $\sigma_{ij} \in H$. Thus, H is a group as desired. □

Proposition 3.3 gives us our desired subgroup of Sym(A). We construct a homomorphism $\varphi : G \to \text{Sym}(A)$, such that $\varphi(G) = \{\sigma_g : g \in G\}$. We refer to φ as the *permutation representation* of the action. When G acts on itself; that is, when $G = A$, $G \cong \varphi(G)$, which is a subgroup of Sym(G).

Proposition 3.4. *Define $\varphi : G \to Sym(A)$ by $g \mapsto \sigma_g$. This function φ is a homomorphism.*

Proof. We show that φ is a homomorphism. Let $g_1, g_2 \in G$. We have that:

$$\begin{aligned}
&\phi(g_1g_2)(a) \\
&= \sigma_{g_1g_2}(a) \\
&= g_1g_2 \cdot a \\
&= g_1 \cdot (g_2 \cdot a) \\
&= \sigma_{g_1}(\sigma_2(a)) \\
&= \phi(g_1)\phi(g_2)(a).
\end{aligned}$$

So ϕ is a homomorphism. □

We conclude by showing $G \cong \varphi(G)$, when G acts on itself by left multiplication.

Proposition 3.5. *Suppose G acts on itself by left multiplication, and let $\varphi : G \to Sym(G)$ be the corresponding permutation representation. Then $G \cong \varphi(G)$.*

Proof. The proof of Proposition 3.4 provides that ϕ is a homomorphism, which is surjective onto $\varphi(G)$. It suffices to show φ is injective. Let $g, h \in G$ such that $\varphi(g) = \varphi(h)$. So $\sigma_g = \sigma_h$, which implies that the permutations agree on all points in G. In particular, $\sigma_g(1) = \sigma_h(1) = g1 = h1 = g = h$. So φ is injective, and we conclude $G \cong \varphi(G)$. □

This concludes the proof of Cayley's Theorem. □

It turns out that it is not necessary for G to act on itself by left multiplication in order for the permutation representation of the action to be isomorphic to G. To this end, we introduce the notion of the kernel and a faithful action.

Definition 89 (Kernel of the Action)**.** Suppose G acts on a set A. The *kernel* of the action is defined as $\{g \in G : g \cdot a = a, \text{ for all } a \in A\}$.

Definition 90 (Faithful Action)**.** Suppose G acts on the set A. The action is said to be *faithful* if the kernel of the action is $\{1\}$.

In particular, if the action is faithful, then each permutation σ_g is unique. So $G \cong \varphi(G)$, where φ is the permutation representation of the action.

One application of group actions is a nice, combinatorial proof of Fermat's Little Theorem. We have already given this proof with Theorem 1.14, but abstracted away the group action. We offer the same proof using the language of group actions below.

Theorem 3.8 (Fermat's Little Theorem). *Let p be a prime number and let $a \in [p-1]$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Proof. Let Λ be an alphabet of order p. Let $\mathbb{Z}_p \cong \langle (1, 2, \ldots, p) \rangle$ act on Λ^p by cyclic rotation. The *orbit* of a string $\omega \in \Lambda^p$, denoted

$$\mathcal{O}(\omega) = \{g \cdot \omega : g \in \mathbb{Z}_p\},$$

consists of either a single string or p strings. Each orbit is an equivalence class under the action. There are a orbits with a single string, where each string is simply a single character repeated p times. The remaining $a^p - a$ strings are partitioned into orbits containing p strings. So $p \big| (a^p - a)$, which implies $a^p \equiv a \pmod{p}$. This is equivalent to $a^{p-1} \equiv 1 \pmod{p}$. □

Lagrange's Theorem is similarly proven. In fact, Fermat's Little Theorem is a special case of Lagrange's Theorem.

Theorem 3.9 (Lagrange's Theorem). *Let G be a finite group, and let $H \leq G$. Then the order of H divides the order of G.*

Proof. Let H act on G by left multiplication. Now fix $g \in G$. We show that the orbit of g,

$$\mathcal{O}(g) = \{h \cdot g : h \in H\},$$

has size $|H|$. We establish a bijection $\varphi : H \to \mathcal{O}(g)$, sending $h \mapsto h \cdot g$. By definition of $\mathcal{O}(g)$, φ is surjective. It suffices to show that φ is injective. Let $h_1, h_2 \in H$ such that $\varphi(h_1) = \varphi(h_2)$. So $h_1 \cdot g = h_2 \cdot g$. By cancellation, $h_1 = h_2$. So φ is injective, as desired. We conclude that $|\mathcal{O}(g)| = |H|$. As g was arbitrary, it follows that the elements of G are partitioned into orbits of order $|H|$. Thus, $|H|$ divides $|G|$, as desired. □

3.1.6 Algebraic Graph Theory- Cayley Graphs

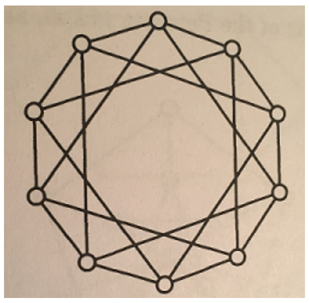
We introduce the notion of a Cayley Graph, which provides an intuitive approach to visualizing the structure of a group. Formally, we define the Cayley Graph as follows.

Definition 91 (Cayley Graph). Let G be a group and let $S \subset G$ such that $1 \not\in S$ and for every $x \in S$, $x^{-1} \in C$. The *Cayley Graph* with respect to G and S is denoted $\text{Cay}(G, S)$ where the vertex set of $\text{Cay}(G, S)$ is G. Two elements $g, h \in G$ are adjacent in $\text{Cay}(G, S)$ if there exists $s \in S$ such that $gs = h$. We refer to S as the *Cayley set*.

We begin with an example of a Cayley graph- the Cycle graph.

Example 97. Let $n \geq 3$, and let $G = \mathbb{Z}_n$ under the operation of addition. Let $S = \{\pm\overline{1}\}$. So the vertices of $\text{Cay}(G, S)$ are the congruence classes $\overline{0}, \overline{1}, \ldots, \overline{n-1}$. We have an edge $\overline{ij}$ if and only if $\overline{j} - \overline{i} = \overline{1}$ or $\overline{j} - \overline{i} = \overline{n-1}$.

Example 98. The Cycle graph is in particular an *undirected circulant graph.* Let $G = \mathbb{Z}_n$, where the operation is addition. Let $S \subset \mathbb{Z}_n$ such that $0 \not\in S$ and $\overline{x} \in S \implies -\overline{x} \in S$. The Cayley graph $\text{Cay}(G, S)$ is an undirected circulant graph. The complete graph K_n is a Cayley graph with $G = \mathbb{Z}_n$ and $S = \mathbb{Z}_n \setminus \{\overline{0}\}$. Similarly, the empty graph is a Cayley graph with $G = \mathbb{Z}_n$ and $S = \emptyset$. We illustrate below the case where $G = \mathbb{Z}_{10}$ with $S = \{\pm\overline{1}, \pm\overline{3}\}$.



We now develop some theory about Cayley Graphs. The first theorem establishes that Cayley graphs are vertex transitive; that is, for every u, v in the group G, there exists an automorphism φ of the Cayley graph mapping $\varphi(u) = v$. The key idea is that G acts transitively on itself by left multiplication. This action induces a transitive action on the Cayley graph.

Theorem 3.10. *Let G be a group with $S \subset G$ as the Cayley set. Let Cay(G, S) be the associated Cayley graph. For any $u, v \in G$, there exists a $\phi \in$ Aut(Cay(G, S)) such that $\phi(u) = v$. (That is, every Cayley graph is vertex transitive).*

Proof. Let $u, v \in G$ be fixed. As G is a group, there exists a unique $g \in G$ such that $gu = v$. Let $\varphi_g : G \to G$ be the function mapping $\varphi_g(u) = gu$. Clearly, $\varphi_g(u) = v$, as desired. The proof of Cayley's Theorem provides that φ_g is an permutation of G. So it suffices to show that φ_g induces a graph homomorphism on $\text{Cay}(G, S)$. Let $x, y \in G$ be adjacent in $\text{Cay}(G, S)$. So there exists a unique $s \in S$ such that $xs = y$. Now $\varphi_g(x) = gx$ and $\varphi_g(y) = gy = gxs = \varphi_g(x)s$. So $s \in S$ still satisfies $\varphi_g(x)s = \varphi_g(y)$. Thus, φ_g induces a graph homomorphism on $\text{Cay}(G, S)$. We conclude that $\varphi_g \in \text{Aut}(\text{Cay}(G, S))$. □

In order for a graph to be vertex-transitive, it is necessary for the graph to be regular. The next lemma shows that a Caley graph $\text{Cay}(G, S)$ is in fact $|S|$-regular.

Lemma 3.3. *Let G be a group and with $S \subset G$ as the Cayley set. Let Cay(G, S) be the associated Cayley graph. Then Cay(G, S) is $|S|$-regular.*

Proof. Let $g \in G$. The $|S|$ neighbors of g are of the form gs, for $s \in S$. As our choice of g was arbitrary, we conclude that $\text{Cay}(G, S)$ is $|S|$-regular. □

We next show that a Cayley graph over a finite group is connected if and only if its Cayley set generates the entire group. The idea is to think of each vertex in the path as a multiplication. So the edge xy in $\text{Cay}(G, S)$ is a multiplication by yx^{-1}. And so a path is a sequence of these multiplications, where the inverses in the interior of the expression cancel. Formally, we have the following.

Theorem 3.11. *Let G be a finite group and let S be the Cayley set. The Cayley graph Cay(G, S) is connected if and only if $\langle S \rangle = G$.*

Proof. Suppose first that $\text{Cay}(G, S)$ is connected. Define $x_1 := 1$, and let $x_k \in G$. As $\text{Cay}(G, S)$ is connected, there exists a path from x_1 to x_k in G. Let $x_1x_2 \dots x_k$ be a shortest path from x_1 to x_k in G. By definition of the Cayley graph, $x_i^{-1}x_{i-1} \in S$ for each $i \in \{2, \dots, k\}$. Applying the multiplications: $(x_k^{-1}x_{k-1})(x_{k-1}^{-1}x_{k-2}) \dots (x_2^{-1}x_1) = x_k^{-1} \in \langle S \rangle$. As x_k was arbitrary, it follows that S generates G.

We now show by contrapositive that $\langle S \rangle = G$ implies $\text{Cay}(G, S)$ is connected. Suppose $\text{Cay}(G, S)$ is not connected. Let $u, v \in \text{Cay}(G, S)$ such that no $u - v$ path exists. Let y be the unique solution to $uy = v$. Then $y \notin \langle S \rangle$, so $\langle S \rangle \neq G$. □

We conclude by providing a vertex-transitive graph that is not a Cayley graph- namely, the Petersen graph.

Theorem 3.12. *The Petersen Graph is not a Cayley Graph.*

Proof. Suppose to the contrary that the Petersen graph is a Cayley graph. There are two groups of order 10: $\mathbb{Z}_{10}$ and D_{10}. As the Petersen graph is 3-regular, we have that a Cayley set $S \subset G$ has three elements. So either one or all three elements are their own inverses. We consider the following cases.

- **Case 1:** Suppose $G = \mathbb{Z}_{10}$. Then the Cayley set $S = \{\overline{a}, -\overline{a}, \overline{5}\}$. Observe that $(\overline{0}, \overline{a}, \overline{5 + a}, \overline{5})$ forms a sequence of vertices that constitute a 4-cycle in $\text{Cay}(\mathbb{Z}_{10}, S)$. However, any pair of non-adjacent vertices in the Petersen graph share precisely one common neighbor, so the Petersen graph has no four-cycle. So the Petersen graph is not the Cayley graph of $\mathbb{Z}_{10}$.
- **Case 2:** Now suppose instead that $G = D_{10}$. As the Petersen graph is connected, S necessarily generates D_{10}. So S necessarily contains an element of the form sr^i for some i. If S has precisely one element of order 2, then $S = \{r^i, r^{-i}, sr^j\}$ for some $i, j \in [4]$. In this case, 1 is adjacent to both r^i and sr^j in $\text{Cay}(D_{10}, S)$. However, r^i and sr^j are both adjacent to sr^{j+i}, creating a four-cycle consisting of $(1, r^i, sr^{j+i}, sr^j)$, a contradiction.

Suppose instead $S = \{sr^i, sr^j, sr^k\}$ for distinct $i, j, k \in \{0, 1, 2, 3, 4\}$. We have 1 is adjacent to each element of S in $\text{Cay}(D_{10}, S)$. The other two neighbors of sr^j are r^{j-i} and r^{j-k}. We next show that sr^k is also adjacent to r^{j-k}. Observe that:

$$\begin{aligned} r^{j-k} \cdot sr^j &= r^{j-k} \cdot r^{-j} s \\ &= r^{-k} s \\ &= sr^k. \end{aligned}$$

So $sr^j \in S$ satisfies $r^{j-k} \cdot sr^j = sr^k$, as desired. So $\text{Cay}(D_{10}, S)$ has a four-cycle of the form $(1, sr^j, r^{j-k}, sr^k)$. In this case, $\text{Cay}(D_{10}, S)$ is not isomorphic to the Petersen graph.

As we have exhausted all possibilities, we conclude that the Petersen graph is not a Cayley graph. □

3.1.7 Algebraic Graph Theory- Transposition Graphs

We now provide some exposition on transpositions. A permutation of $[n]$ can be viewed as a directed graph with vertex set $[n]$, which is the disjoint union of directed cycles. Each directed cycle in the grpah corresponds to a cycle in the permutation's cycle decomposition. Furthermore, each permutation cycle can be decomposed as the product of transpositions, or 2-cycles. The transpositions are viewed as edges. We adapt this framing to study the Symmetry group from an algebraic standpoint.

Formally, let $\mathcal{T}$ be a set of transpositions. We define the graph T with vertex set $[n]$ and edge set

$$E(T) = \{ij : (ij) \in \mathcal{T}\}.$$

We say that $\mathcal{T}$ is *generating set* if $\text{Sym}(n) = \langle \mathcal{T} \rangle$, and $\mathcal{T}$ is *minimal* if for any $g \in \mathcal{T}$, $\mathcal{T} \setminus \{g\}$ is not a generating set. Note that T is not a Cayley graph, but it is useful in studying the Cayley graphs of $\text{Sym}(n)$.

We begin with an analogous result to Theorem 3.11, for T rather than Cayley Graphs.

Lemma 3.4. *Let $\mathcal{T}$ be a set of transpositions from Sym(n). Then $\mathcal{T}$ is a generating set for Sym(n) if and only if its graph T is connected.*

Proof. Let T be the graph of $\mathcal{T}$. Suppose $(1i), (ij) \in \mathcal{T}$. Then:

$$(ij)(1i)(ij) = (1j) \in \langle \mathcal{T} \rangle.$$

By induction, if there exists a $1 - k$ path in T, then $(1k) \in \langle \mathcal{T} \rangle$. It follows that for any x, y on the same component, then $(xy) \in \langle \mathcal{T} \rangle$. So the transpositions belonging to a certain component generate the symmetric group on the vertices of that component. Thus, if T is connected, then $S_n = \langle \mathcal{T} \rangle$.

We next show that if $\langle \mathcal{T} \rangle = \text{Sym}(n)$, then T is connected. This will be done by contrapositive. If T is not connected, then no transposition of $\mathcal{T}$ can map a vertex from one component to the other. □

Remark: The components of T are precisely the orbits of $\langle T \rangle$ acting on $[n]$.

Lemma 3.4 is quite powerful. It implies that every minimal generating set $\mathcal{T}$ of transpositions has the same cardinality. In particular, the graph of any minimal generating set is a spanning tree, so every minimal generating set has $n - 1$ transpositions. This allows us to answer the following questions.

1. Is a set of transpositions $\mathcal{T}$ of S_n a generating set?
2. Is a generating set of transpositions $\mathcal{T}$ of S_n minimal?
3. If a set of transpositions is a generating set, which transpositions can be removed while still generating S_n?
4. If a set of transpositions is not a generating set, which transpositions are missing?

In order to answer these questions, we reduce to the spanning tree problem and the connectivity problem. Question 1 is answered by Lemma 3.4- we simply check if the graph T corresponding to $\mathcal{T}$ is connected, which can be done using Tarjan's algorithm which runs in $\mathcal{O}(|V| + |E|)$ time. In order to answer Question 2, Lemma 3.4 implies that it suffices to check if T is a spanning tree. So first, we first check if the graph T is connected. If so, it suffices to check if T has $n - 1$ edges, as that is the characterization of a tree.

Using our theory of spanning trees, we easily answer Question 3 as well. We can construct a spanning tree by removing an edge from a cycle, then applying the procedure recursively to the subgraph. As the transpositions of $\mathcal{T}$ correspond to edges of T, this fact about spanning trees allows us to remove transpositions from $\mathcal{T}$ while allowing the modified $\mathcal{T}$ to generate $\text{Sym}(n)$.

Finally, to answer Question 4, we simply select pairs of vertices from two components of T and add an edge $e = ij$, which corresponds to setting $\mathcal{T} := \mathcal{T} \cup \{(ij)\}$. We repeat the procedure for the modified $\mathcal{T}$ until it is connected.

We conclude with a final lemma, which relates the graph T for a set of transpositions $\mathcal{T}$ to the Cayley graph $\text{Cay}(\text{Sym}(n), \mathcal{T})$.

Lemma 3.5. *Let $\mathcal{T}$ be a non-empty set of transpositions from Sym(n), and let $g, h \in \mathcal{T}$. Suppose that the graph T of $\mathcal{T}$ contains no triangles. If $gh \neq hg$, then g and h have exactly one common neighbor in the Cayley graph Cay(Sym(n), $\mathcal{T}$). Otherwise, g and h have exactly two common neigbors in Cay(Sym(n), $\mathcal{T}$).*

Proof. The neighbors of g in $\text{Cay}(\text{Sym}(n), \mathcal{T})$ are of the form gx, where $x \in \mathcal{T}$. In particular, if g, h have a common neighbor in $\text{Cay}(\text{Sym}(n), \mathcal{T})$, then there exist $x, y \in \mathcal{T}$ satisfying $gx = hy$.

Suppose that $gh = hg$. Then g and h have disjoint support. So $gh = hg$ is a common neighbor of g, h. As g, h are transpositions, $g^2 = h^2 = 1$, so g, h have a common neighbor of (1). These are precisely the two common neighbors of g, h in $\text{Cay}(\text{Sym}(n), \mathcal{T})$.

Suppose instead $hg \neq gh$. Then g, h are not disjoint. Without loss of generality, suppose $g = (1, 3)$ and $h = (1, 2)$. Then $hg = (1, 2, 3)$, which has three factorizations: $(1, 2)(1, 3) = (1, 3)(2, 3) = (2, 3)(1, 2)$. Note that if $(2, 3) \in \mathcal{T}$, then the vertices $1, 2, 3$ induce a triangle in T, the graph of $\mathcal{T}$. Thus, $(2, 3) \notin \mathcal{T}$. So $(1, 2)(1, 3)$ is the unique factorization of hg in $\mathcal{T}$, yielding (1) as the unique neighbor of g, h. □

3.2 Subgroups

One basic approach in studying the structure of a mathematical satisfying a set of axioms is to study subsets of the given object which satisfy the same axioms. A second basic method is to collapse a mathematical object onto a smaller object sharing the same structure. This collapsed structure is known as a *quotient*. Both of these themes recur in algebra: in group theory with subgroups and quotient groups; in ring theory with subrings and quotient rings; in linear algebra with subspaces and quotient spaces of vector spaces; etc. A clear understanding of subgroups is required to study quotient groups, with the notion of a *normal subgroup*.

Definition 92 (Subgroup)**.** Let G be a group, and let $H \subset G$. H is said to be a *subgroup* of G if H is also a group. We denote the subgroup relation as $H \leq G$.

We begin with some examples of subgroups.

Example 99. $\mathbb{Z} \leq \mathbb{Q}$ and $\mathbb{Q} \leq \mathbb{R}$ with the operation of addition.

Example 100. The group of rotations $\langle r \rangle \leq D_{2n}$

Example 101. $D_{2n} \leq S_n$

Example 102. The set of even integers is a subgroup of $\mathbb{Z}$ with the operation of adddition.

We begin with the subgroup test, which allows us to verify a subset H of a group G is actually a subgroup without verifying the group axioms.

Proposition 3.6 (The Subgroup Criterion)**.** *Let G be a group, and let $H \subset G$. $H \leq G$ if and only if:*

1. $H \neq \emptyset$

2. For all $x, y \in H$, $xy^{-1} \in H$

Furthermore, if H is finite, then it suffices to check that H is non-empty and closed under multiplication.

Proof. If $H \leq G$, then conditions (1) and (2) follow immediately from the definition of a group. Conversely, suppose that H satisfies (1) and (2). Let $x \in H$ (such an x exists because H is non-empty). As H satisfies condition (2), we let $y = x$ to deduce that $xx^{-1} = 1 \in H$. As H contains the identity of G, we apply property (2) to obtain that $1x^{-1} = x^{-1} \in H$ for every $x \in H$. So H is closed under inverses. We next show that H is closed under product. Let $x, y^{-1} \in H$. Then by property (2) and the fact that $(y^{-1})^{-1} = y$, $xy \in H$. As the operation of G is associative, we conclude that H is a subgroup of G.

Now suppose that H is finite and closed under multiplication. Let $x \in H$. As H is closed under multiplication, $\langle x \rangle \subset H$. As H is finite, $x^{-1} \in \langle x \rangle$. So H is closed under inverses and $H \leq G$. □

Example 103. We use the subgroup criterion to verify that the set of even integers is a subgroup of $\mathbb{Z}$ over addition. We have that 0 is an even integer. Now let $2x, 2y$ be even integers where $x, y \in \mathbb{Z}$. We have $(2y)^{-1} = -2y$, so $2x(2y)^{-1} = 2x - 2y = 2(x - y)$. As $\mathbb{Z}$ is closed under addition and inverses, $x - y \in \mathbb{Z}$. So $2(x - y)$ is an even integer.

We explore several families of subgroups, which yield many important examples and insights in the study of group theory. Two important problems in group theory include studying (a) how "far away" a group is from being commutative; and (b) in a group homomorphism $\phi : G \to H$, which members of G map to 1_H? On the surface, these problems do not appear to be related. In fact, both these problems are closely related. We examine a specific class of group action known as the *action of conjugation*. Studying the kernels and stabilizers of these actions provide invaluable insights about the level of commutativity of for the given group. We begin by studying the commutativity of a group, with the centralizer, normalizer, and center of a group.

Definition 93 (Centralizer)**.** Let G be a group and let $A \subset G$ be non-empty. The *centralizer* of G is the set: $C_G(A) = \{g : ga = ag, \text{ for all } a \in A\}$. That is, $C_G(A)$ is the set of elements of G which commute with every element of A.

Remark: It is common to write $C_G(A) = \{g \in G : gag^{-1} = a, \text{ for all } a \in A\}$, which is equivalent to what is presented in the definition. We see the notation gag^{-1} again when discussing the normalizer, and more generally when discussing the action of conjugation. By convention, when $A = \{a\}$, we write $C_G(\{a\}) = C_G(a)$.

Proposition 3.7. *Let G be a group, and let A be a non-empty subset of G. Then $C_G(A) \leq G$.*

Proof. We appeal to the subgroup criterion. Clearly, $1 \in C_G(A)$, so $C_G(A) \neq \emptyset$. Now suppose $x, y \in C_G(A)$ and let $a \in A$. It follows that $xya = xay = axy$, as x, y commute with a. So $C_G(A)$ is closed under the group operation. Finally, if $g \in C_G(A)$ and $a \in A$, we have $gag^{-1} = a$, which is equivalent to $ag^{-1} = g^{-1}a$, so $C_G(A)$ is closed under inverses. So $C_G(A) \leq G$. □

Example 104. Let G be a group and let $a \in G$. Then $\langle a \rangle \leq C_G(a)$, as powers of a commute with a by associativity of the group operation.

Example 105. Let G be an Abelian group. Then $C_G(A) = G$ for any non-empty $A \subset G$.

Example 106. Recall Q_8, the Quaternion group of order 8. By inspection, we see that $C_{Q_8}(i) = \{\pm 1, \pm i\}$. Observe that $ij = k$ while $ji = -k$, so $j \not\in C_{Q_8}(i)$. If we consider $-j$ instead, we see that $-ji = k$ while $i(-j) = k$, so $-j \not\in C_{Q_8}(i)$. By similar argument, it can be verified that $\pm k \not\in C_{Q_8}(i)$.

We could alternatively use Lagrange's Theorem to compute $C_{Q_8}(i)$. Recall that Lagrange's Theorem states that $|C_{Q_8}(i)|$ divides $|Q_8| = 8$. As $\langle i \rangle \leq C_{Q_8}(i)$, we have $|C_{Q_8}(i)| \in \{4, 8\}$. As $j \not\in C_{Q_8}(i)$, $|C_{Q_8}(i)| \neq 8$. Therefore, $C_{Q_8}(i) = \langle i \rangle$.

We next introduce the notion of the *center* of a group, which is a special case of the centralizer.

Definition 94 (Center)**.** Let G be a group. The *center* of G is the set $Z(G) = \{g \in G : gx = xg, \text{ for all } x \in G\}$. That is, the center is the set of elements in G which commute with every element in G.

Remark: Observe that $Z(G) = C_G(G)$, so $Z(G) \leq G$. We also clearly have:

$$Z(G) = \bigcap_{g \in G} C_G(g).$$

The next subgroup we introduce is the normalizer, which is a generalization of the centralizer. Intuitively, the centralizer is the set of elements that commute with a non-empty $A \subset G$. However, the normalizer simply preserves the set A under this notion of conjugation. That is, if g is in the normalizer of A, then $x \in A$, there exists a $y \in A$ such that $gx = yg$. So the elements of A may map to each other rather than preserved by commutativity. The normalizer is formalized as follows.

Definition 95 (Normalizer)**.** Let G be a group, and let $A \subset G$. The *normalizer* of A with respect to G is the set $N_G(A) = \{g \in G : gAg^{-1} = A\}$.

Clearly, $C_G(A) \leq N_G(A)$ for any non-empty $A \subset G$. We now compute the centralizer, center, and normalizer for D_8.

Example 107. If G is Abelian, $Z(G) = C_G(A) = N_G(A) = G$ for any non-empty $A \subset G$.

Example 108. Let $G = D_8$ and let $A = \langle r \rangle$. Clearly, $A \leq C_G(A)$. As $sr = r^{-1}s \neq rs$, $s \not\in C_G(A)$. Now suppose some $sr^i \in C_G(A)$. Then $sr^ir^{-i} = s \in C_G(A)$, a contradiction. So $C_G(A) = A$.

Example 109. $N_{D_8}(\langle r \rangle) = D_8$. We consider:

$$s\langle r \rangle s = \{s1s, srs, sr^2s, sr^3s\} = \{1, r^{-1}, r^2, r^{-3}\}$$

As $N_{D_8}(\langle r \rangle)$ is a group, s is multiplied by each rotation. So we obtain $N_{D_8}(\langle r \rangle) = D_8$.

Example 110. $Z(D_8) = \{1, r^2\}$. As $Z(D_8) \leq C_{D_8}(\langle r \rangle)$, it suffices to show r and r^3 do not commute with some element of D_8. We have $rs = sr^{-1}$ by the presentation of D_8. Similarly, $r^3s = sr^{-3}$. So $Z(D_8) = \{1, r^2\}$.

We next introduce the stabilizer of a group action, which is a special subgroup which contains elements of G that fix a specific element $a \in A$.

Definition 96 (Stabilizer)**.** Let G act on the set A. For each $a \in A$, the stabilizer $\text{Stab}(a) = \{g \in G : g \cdot a = a\}$.

Remark: Clearly, the Kernel of the action is simply:

$$\bigcap_{a \in A} \text{Stab}(a),$$

which contains the set of all group elements g that fix every point in A.

We defined the kernel in the previous section on group actions. More generally, we define the kernel of a homomorphism as follows.

Definition 97 (Kernel of a Homomorhism)**.** Let $\phi : G \to H$ be a group homomorphism. We denote the *kernel* of the homomorphism ϕ as $\ker(\phi) = \phi^{-1}(1_H)$, or the set of elements in G which map to 1_H under ϕ.

Remark: Recall that the Kernel of a group action is the set of group elements which fix every element of A. We equivalently define the Kernel of a group action as $\ker(\phi) = \phi^{-1}((1))$, where $\phi : G \to S_A$ is the homomorphism defined in Cayley's theorem sending $g \mapsto \sigma_g$, a permutation. Both the Kernel and the Stabilizers are subgroups of G.

We now explore the relation between normalizers, centralizers, and centers, and the kernels and stabliziers of group actions. In particular, normalizers, and centers of groups are stabilizers of group actions. We begin with the action of conjugation.

Definition 98 (Action of Conjugation)**.** Let G be a group, and let A be a set. G acts on A by conjugation, by mapping $(g, a) \in G \times A$ to gag^{-1}.

Proposition 3.8. *Suppose G acts on 2^G by conjugation. Then $N_G(A) = G_A$, the stabilizer of A.*

Proof. Let $A \in 2^G$. Let $g \in G_A$. Then $gAg^{-1} = A$, so $g \in N_G(A)$ and $G_A \subset N_G(A)$. Conversely, let $h \in N_G(A)$. By definition of the normalizer, $hAh^{-1} = A$, so h fixes A. Thus, $h \in G_A$ and $N_G(A) \subset G_A$. $\square$

Remark: It follows that the kernel of the action of G on 2^G by conjugation is:

$$\bigcap_{A \subset G} N_G(A).$$

By similar analysis, we consider the action of $N_G(A)$ on the set A by conjugation. So for $g \in G$, we have:

$$g : a \mapsto gag^{-1}.$$

By definition of $N_G(A)$, this maps $A \to A$. We observe that $C_G(A)$ is the kernel of this action. It follows from this that $C_G(A) \leq N_G(A)$. A little less obvious is that $Z(G)$ is the kernel of G acting on itself by conjugation.

Proposition 3.9. *Let G act on itself by conjugation. The kernel of this action Ker $= Z(G)$.*

Proof. Let $g \in \text{Ker}$, and let $h \in G$. Then $ghg^{-1} = h$ by definition of the Kernel. So $gh = hg$, and $g \in Z(G)$. So $\text{Ker} \subset Z(G)$. Conversely, let $x \in Z(G)$. Then $xh = hx$ for all $h \in G$. So $xhx^{-1} = h$ for all $x \in \text{Ker}$ and $Z(G) \subset \text{Ker}$. $\square$

3.2.1 Cyclic Groups

In this section, we study cyclic groups, which are generated by a single element. The results in this section are number theoretic in nature. There is relatively little meat in this section, but the results are quite important for later. So it is important to spell out certain details. We begin with the definition of a cyclic group below.

Definition 99 (Cyclic Group)**.** A group G is said to be cyclic if $G = \langle x \rangle = \{x^n : n \in \mathbb{Z}\}$ for some $x \in G$.

Remark: As the elements of G are of the form x^n, associativity, closure under multiplication, and closure under inverses follows immediately. We have that $x^0 = 1$, so $G = \langle x \rangle$ is a group.

Recall that the order of an element x in a group is the least positive integer n such that $x^n = 1$. Equivocally, $|x| = |\langle x \rangle|$. We formalize this as follows.

Proposition 3.10. *If $H = \langle x \rangle$, then $|H| = |x|$. More specificially:*

1. *If $|H| = n < \infty$, then $x^n = 1$ and $1, x, \dots, x^{n-1}$ are all distinct elements of H; and*
2. *If $|H| = \infty$, then $x^n \neq 1$ for all $n \neq 0$; and $x^a \neq x^b$ for all $a \neq b \in \mathbb{Z}$.*

Proof. Suppose first that $|x| = n < \infty$. Let $a, b \in \{0, \dots, n-1\}$ be distinct such that $x^a = x^b$. Then $x^{b-a} = x^0 = 1$, contradicting the fact that n is the minimum integer such that $x^n = 1$. So all the elements of $1, x, \dots, x^{n-1}$ are unique. It suffices to show that $H = \{1, x, \dots, x^{n-1}\}$. Consider x^t. By the Division Algorithm, $x^t = x^{nq+k}$ for some $q \in \mathbb{Z}$ and $k \in \{0, \dots, n-1\}$. Then $x^t = (x^n)^q x^k = 1^q x^k = x^k \in \{1, \dots, x^{n-1}\}$. So $|H| = |x|$.

Now suppose $|x| = \infty$. So no positive power of x is the identity. Now suppose $x^a = x^b$ for distinct integers a, b. Clearly, $x^{a-b} \neq x^0 = 1$; otherwise, we have a positive integer such that $x^n = 1$, a contradiction. So distinct powers of x are distinct elements of H, and we have $|H| = \infty$. $\square$

Remark: Proposition 3.10 allows us to reduce powers of x based on their congruence classes modulo $|x|$. In particular, $\langle x \rangle \cong \mathbb{Z}_n$ when $|x| = n < \infty$. If $n = \infty$, then $\langle x \rangle \cong \mathbb{Z}$. In order to show this, we need a helpful lemma.

Proposition 3.11. *Let G be a group, and let $x \in G$. Suppose $x^m = x^n = 1$ for some $m, n \in \mathbb{Z}^+$. Then for $d = gcd(m, n)$, $x^d = 1$. In particuar, if $x^m = 1$ for some $m \in \mathbb{Z}$, then $|x|$ divides m.*

Proof. By the Euclidean Algorithm, we write $d = mr + ns$, for appropriately chosen integers r, s. So $x^d = x^{mr+ns} = (x^m)^r \cdot (x^n)^s = 1$.

We now show that if $x^m = 1$, then $|x|$ divides m. If $m = 0$, then we are done. Now suppose $m \neq 0$. We take $d = \gcd(m, |x|)$. By the above argument, we have that $x^d = 1$. As $1 \leq d \leq |x|$ and $|x|$ is the least such positive integer k that $x^k = 1$, it follows that $d = |x|$. As $d = \gcd(m, |x|)$, it follows that $d = |x|$ divides m. $\square$

We now show that every cyclic group is isomorphic to either the integers or the integers modulo n, for some n. We first introduce the notion of a well-defined function, which we need for this next theorem.

Definition 100 (Well-Defined Function)**.** A map $\phi : X \to Y$ is well-defined if for every x, there exists a unique y such that $\phi(x) = y$. In particular, if X is a set of equivalence classes, then for any two a, b in the same equivalence class, $\phi(a) = \phi(b)$.

Theorem 3.13. *Any two cyclic groups of the same order are isomorphic. In particular, we have the following.*

1. *If $|\langle x\rangle| = |\langle y\rangle| = n < \infty$, then the map: $\phi : \langle x\rangle \to \langle y\rangle$ sending $x^k \mapsto y^k$ is a well-defined isomorphism.*
2. *If $\langle x\rangle$ has infinite order, then the map $\psi : \mathbb{Z} \to \langle x\rangle$ sending $k \mapsto x^k$ is a well-defined isomorphism.*

Proof. Let $\langle x\rangle, \langle y\rangle$ be cyclic groups of finite order n. We show that $x^k \mapsto y^k$ is a well-defined isomorphism. Let r, s be distinct positive integers such that $x^r = x^s$. In order for ϕ to be well-defined, it is necessary that $\phi(x^r) = \phi(x^s)$. As $x^{r-s} = 1$, we have by Proposition 3.11 that n divides $r - s$. So $x^r = tn + s$. It follows that:

$$\begin{aligned}\phi(x^r) &= \phi(x^{tn+s}) \\ &= y^{tn+s} \\ &= (y^n)^t y^s \\ &= y^s = \phi(x^s).\end{aligned}$$

So ϕ is well-defined. By the laws of exponents, we have:

$$\begin{aligned}\phi(x^a x^b) &= y^{ab} \\ &= y^a y^b \\ &= \phi(x^a)\phi(x^b).\end{aligned}$$

So ϕ is a homomorphism. It follows that since y^k is the image of x^k under ϕ, that ϕ is surjective. As $|\langle x\rangle| = |\langle y\rangle| = n$, ϕ is injective. So ϕ is a homomorphism.

Now suppose $\langle x\rangle$ is infinite. We have from Proposition 3.10 that for any two distinct integers a, b that $x^a \neq x^b$. So ψ is well-defined and injective. It follows immediately from the rules of exponents that ψ is a homomorphism. It suffices to show ψ is surjective. Let $h \in \langle x\rangle$. Then $h = x^k$ for some $k \in \mathbb{Z}$. So k is the preimage of h under ψ, and we have ψ is surjective. So ψ is an isomorphism. $\square$

We conclude this section with some additional results that are straight-forward to prove. This first proposition provides results for selecting generators of a cyclic group.

Proposition 3.12. *Let $H = \langle x\rangle$. If $|x| = \infty$, then $H = \langle x^a\rangle$ if and only if $a = \pm 1$. If $|x| = n < \infty$, then $H = \langle x^a\rangle$ if and only if gcd$(a, n) = 1$.*

Proof. Suppose that $|H| = \infty$. If $a = \pm 1$, then $H = \langle x^a\rangle$. Conversely, let $a \in \mathbb{Z}$ such that $H = \langle x^a\rangle$. If $a = \pm 1$, then we are done. So suppose to the contrary that there a is an integer other than ± 1 such that $H = \langle x^a\rangle$. Without loss of generality, suppose $a > 0$. Let $b \in \{-a+1, \ldots, -1, 1, \ldots, a-1\}$. No such integer k exists such that $ak = b$. So it is necessary that $a = \pm 1$.

We now consider the case in which $|H| = n < \infty$. We have $H = \langle x^a\rangle$ if and only if $|x^a| = |x|$. This occurs if and only if $|x^a| = \dfrac{n}{\gcd(n, a)} = n$, which is equivalent to $\gcd(a, n) = 1$. The Euler ϕ function counts the number of integers relatively prime to the input, so there are $\phi(n)$ members of H which individually generate H. $\square$

We conclude this section with the following result.

Theorem 3.14. *Let $G = \langle x \rangle$. Then every subgroup of G is also cyclic.*

Proof. Let $H \leq G$. If $H = \{1\}$, we are done. Suppose $H \neq \{1\}$. Then there exists an element $x^a \in H$ where $a > 0$ (if we selected x^a with $a < 0$, then we obtain $x^{-a} \in H$ as H is closed under inverses, and so $-a > 0$). By the Well-Ordering Principle, there exists a least positive b such that $x^b \in H$. Clearly, $\langle x^b \rangle \leq H$. Now let $x^a \in H$. Then by the Division Algorithm, $x^a = x^{kb+r}$ for $k \in \mathbb{Z}$ and $0 \leq r < b$. So $x^r = x^a(x^b)^{-k}$. As $x^a, x^b \in H$, so is x^r. But since b is the least positive integer such that $x^b \in H$, then $r = 0$. So $H \leq \langle b \rangle$, and H is cyclic. □

3.2.2 Subgroups Generated By Subsets of a Group

In this section, we generalize the notion of a cyclic group. A cyclic group is generated by a single element. We examine subgroups which are generated by one or more elements of the group, rather than just a single element. The important result in this section is tht subgroups of a group G are closed under intersection.

Theorem 3.15. *Let G be a group, and let $\mathcal{A}$ be a collection of subgroups of G. Then the intersection of all the members of $\mathcal{A}$ is also a subgroup of G.*

Proof. We appeal to the subgroup criterion. Let:

$$K = \bigcap_{H \in \mathcal{A}} H.$$

As each $H \in \mathcal{A}$ is a subgroup of G, $1 \in H$ for each $H \in \mathcal{A}$. So $1_H \in K$ and we have $K \neq \emptyset$. Now let x, y. As $x, y \in H$ for each $H \in \mathcal{A}$, we have xy^{-1} also in each $H \in \mathcal{A}$. So $xy^{-1} \in K$ and we are done. So $K \leq G$. □

We now examine precisely the construction of the subgroup generated by a set $A \subset G$. Formally, we have the proposition.

Proposition 3.13. *Let $A \subset G$. Then:*

$$\langle A \rangle = \bigcap_{\substack{A \subset H \\ H \leq G}} H.$$

Proof. Let $\mathcal{A} = \{H \leq G : A \subset H\}$. As $\langle A \rangle \in \mathcal{A}$, $\mathcal{A} \neq \emptyset$. Let

$$K = \bigcap_{H \in \mathcal{A}} H.$$

Clearly, $A \subset K$. Since $K \leq G$ by Theorem 3.15, $\langle A \rangle \leq K$. As $\langle A \rangle$ is the unique minimal subgroup of G containing A, it follows that $\langle A \rangle \in \mathcal{A}$ and $K \leq \langle A \rangle$. □

3.2.3 Subgroup Poset and Lattice (Hasse) Diagram

The goal of this section is to provide another visual tool for studying the structure of the graph. While the Cayley Graph describes the intuitive notion of spanning of a subset of a group, the lattice (or Hasse) diagram depicts the subgroup relation using a directed graph. The lattice diagram and associated structure known as a poset are quite useful in studying the structure of a group. In the section on quotients, we see immediate benefit when studying the Fourth (or Lattice) Isomorphism Theorem. We begin with the definition of a poset.

Definition 101 (Partially Ordered Set (Poset))**.** A *partially ordered set* or *poset* is a pair $(S, \leq)$, where S is a set and $\leq$ is a binary relation on S satisfying the following properties:

- Reflexivity: $a \leq a$ for all $a \in S$.
- Anti-Symmetry: $a \leq b$ and $b \leq a$ implies that $a = b$.
- Transitivity: $a \leq b$ and $b \leq c$ implies that $a \leq c$.

Intuitively, a partial order behaves like the natural ordering on $\mathbb{Z}$. Consider $3, 4, 5 \in \mathbb{Z}$. We have $3 \leq 3$. More generally, $a \leq a$ for any $a \in \mathbb{Z}$. So reflexivity holds. Transitivity similarly holds, as is illustrated with the example that $3 \leq 4$ and $4 \leq 5$. We have $3 \leq 5$ as well. Anti-symmetry holds as well. We now consider some other examples of posets.

Example 111. The set $\mathbb{N}$ with the relation of divisibility forms a poset. Recall the divisibility relation $a \big| b$ if there exists an integer q such that $aq = b$.

Example 112. Let S be a set. The subset relation $\subset$ is a partial order over 2^S.

Example 113. Let G be a group. Let $\mathcal{G} = \{H : H \leq G\}$. The subgroup relation forms a partial order over $\mathcal{G}$.

We now describe how to construct the Hasse Diagram for a poset. The vertices of the Hasse diagram are the elements of the poset S. There is a directed edge (i, j) if $i \leq j$ and there is no other element k such $i \leq k$ and $k \leq j$. In the poset of subgroups, the trivial subgroup $\{1\}$ is at the root of the Hasse diagram and the group G is at the top of the diagram. Careful placement of the elements of the poset can yield a simple and useful pictoral representation of the structure. A directed path along the Hasse diagram provides information on the transitivity relation. That is, the directed path H, J, K, M indicates that $H \leq J$, $J \leq K$, and $K \leq M$. So H is also a subgroup of K and M; and J is also a subgroup of M.

Let H, K be subgroups of G. We leverage the Hasse Diagram to find $H \cap K$. Additionally, the Hasse Diagram allows us to ascertain the *join* of H and K, denoted $\langle H, K \rangle$, which is the smallest possible subgroup containing H and K. Note that the elements of H and K are multiplied together. So $H \cup K$ is not necessarily the same set as $\langle H, K \rangle$. In fact, $H \cup K$ may not even form a group.

In order to find $H \cap K$, we find H and K in the Hasse Diagram. Then we enumerate all paths from 1 to H, as well as all paths from $1 \to K$. We examine all subgroups M that lie on some $1 \to H$ path and some $1 \to K$ path. The intersection $H \cap K$ is the subgroup M closest to both H and K. Similarly, if we start at H and K and enumerate the paths to G on the Hasse Diagram, the closest reachable subgroup from H and K is $\langle H, K \rangle$.

Ultimately, we are seeking to leverage visual intuition. We consider Hasse diagrams for $\mathbb{Z}_8$. Observe that the cyclic subgroups generated by $\overline{2}$ and $\overline{4}$ are isomorphic to $\mathbb{Z}_4$ and $\mathbb{Z}_2$ respectively. Here, the trivial subgroup $\{\overline{0}\}$ is at the bottom of the lattice diagram and is contained in each of the succeeding subgroups. We then see that $\langle \overline{4} \rangle \leq \langle \overline{2} \rangle$, and in turn that each of these are subgroups of $\mathbb{Z}_8$. If we consider the sublattice from $\{\overline{0}\}$ to $\langle \overline{2} \rangle$, then we have the lattice for $\mathbb{Z}_4$.

Example 114.

$$\begin{tikzcd}
\mathbb{Z}_8 = \langle \overline{1} \rangle \arrow[d, dash] \\
\langle \overline{2} \rangle \cong \mathbb{Z}_4 \arrow[d, dash] \\
\langle \overline{4} \rangle \cong \mathbb{Z}_2 \arrow[d, dash] \\
\langle \overline{8} \rangle = \{\overline{0}\}
\end{tikzcd}$$

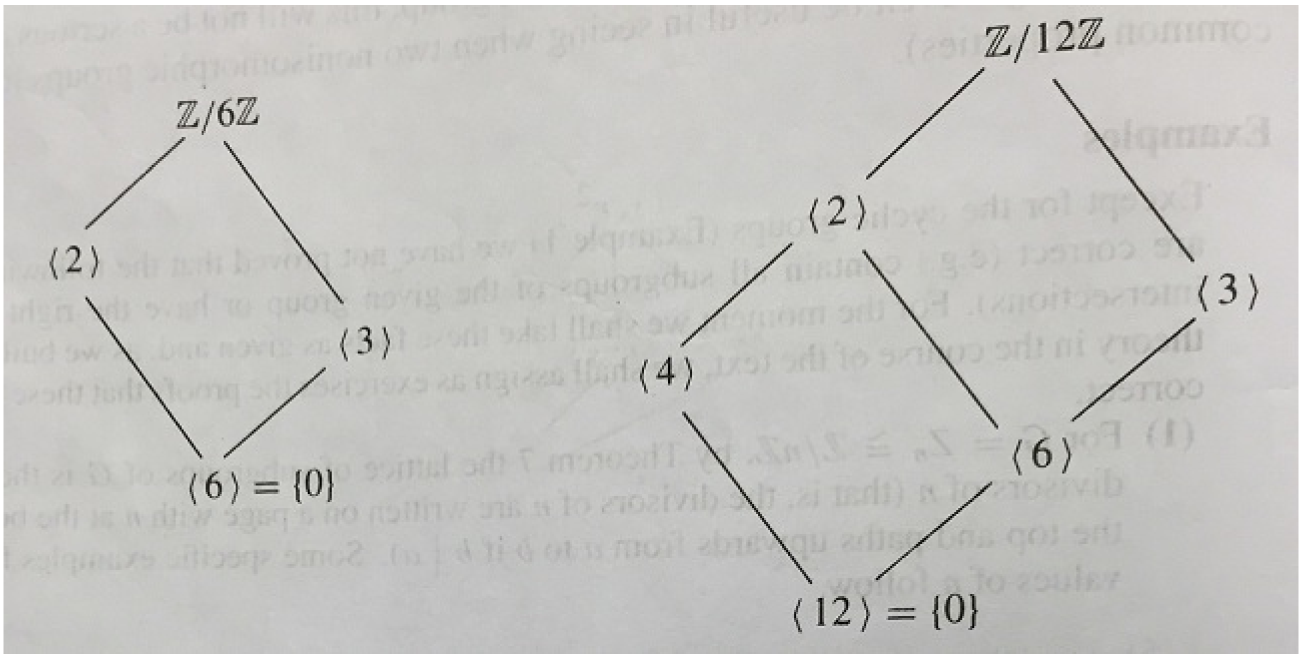
In particular, if p is prime, we see the lattice diagram of $\mathbb{Z}_{p^n}$ is:

Example 115.

$$\begin{tikzcd}
\mathbb{Z}_{p^n} = \langle \overline{1} \rangle \arrow[d, dash] \\
\langle \overline{p} \rangle \arrow[d, dash] \\
\langle \overline{p^2} \rangle \arrow[d, dash] \\
\langle \overline{p^3} \rangle \arrow[d, dash] \\
\vdots \arrow[d, dash] \\
\langle \overline{p^{n-1}} \rangle \arrow[d, dash] \\
\{\overline{0}\}
\end{tikzcd}$$

We now examine the Hasse diagrams of $\mathbb{Z}_6$ and $\mathbb{Z}_{12}$. Observe that in the lattice diagram of $\mathbb{Z}_{12}$, we have $\langle\overline{2}\rangle \cong \mathbb{Z}_6$. Similarly, $\langle\overline{4}\rangle$ in the lattice of $\mathbb{Z}_{12}$ corresponds to $\langle\overline{2}\rangle$ in the lattice of $\mathbb{Z}_6$. Following this pattern, we observe that the lattice of $\mathbb{Z}_6$ can be extracted from the lattice of $\mathbb{Z}_{12}$.

Example 116.



The next group we examine is the Klein group of order 4 (Viergruppe), which is denoted $\mathbb{V}_4$. Formally, $\mathbb{V}_4 \cong \mathbb{Z}_2 \times \mathbb{Z}_2$. So there are three subgroups of order 2 and the trivial subgroup as the precise subgroups of $\mathbb{V}_4$. This yields the following lattice.

Example 117.

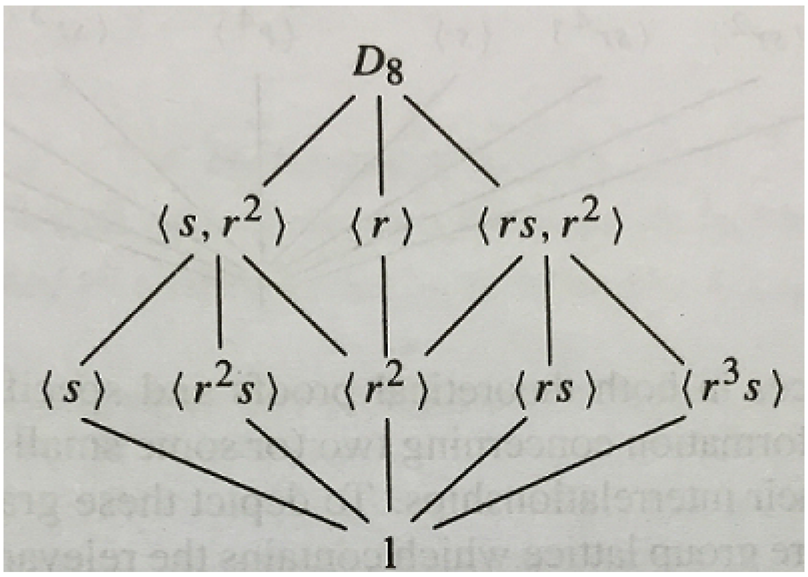
$$\begin{tikzcd}
& \mathbb{V}_4 \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
\langle(1,0)\rangle \arrow[dr, no head] & \langle(0,1)\rangle \arrow[d, no head] & \langle(1,1)\rangle \arrow[dl, no head] \\
& \{(0,0)\} &
\end{tikzcd}$$

In fact, the two distinct groups of order 4 are $\mathbb{Z}_4$ and $\mathbb{V}_4$. It is easy to see that $\mathbb{V}_4 \not\cong \mathbb{Z}_4$ by examining their lattices. It is not true in general that two groups with the same lattice structure are isomorphic. We will also see that $\mathbb{V}_4$ is isomorphic to a subgroup of D_8, and we will leverage the lattice of D_8 to obtain this result.

We now construct the lattice of D_8. Recall that Lagrange's Theorem states that if G is a finite group and $H \leq G$, then $|H|$ divides $|G|$. In $\mathbb{Z}_n$, we see that if q divides n, then $\mathbb{Z}_q \leq \mathbb{Z}_n$. In general, the converse of Lagrange's Theorem is not true. Furthermore, there could be many subgroups of a given order. In D_8, we have subgroups of order $1, 2$, and 4. We begin by enumerating the subgroups of order 2, then taking their joins to obtain all but one of the subgroups of order 4. The remaining subgroup of order four is $\langle r \rangle$, which only has $\langle r^2 \rangle$ as an order 2 subgroup. Then the subgroups of order 4 all have directed edges to D_8 in the lattice. Recall that each reflection, which is of the form sr^i for $i \in \{0, \ldots, 3\}$, has order 2. Similarly, r^2 has order 2 as well. This yields the five subgroups of order 2 which are adjacent to $\{1\}$, the trivial subgroup.

It should be readily apparent that the three subgroups of order 4 specified in the lattice of D_8 below exist. What may not be as obvious is that these are precisely the three subgroups of order 4. There are precisely two distinct groups of order 4: $\mathbb{Z}_4$ and $\mathbb{V}_4$. It is clear that $\langle r \rangle \cong \mathbb{Z}_4$. Now $\mathbb{V}_4$ has three subgroups of order 2. We may check by exhaustion the joins of all $\binom{5}{3} = 10$ sets of three subgroups of order 2. The join of any three subgroups of order two which allows us to isolate r or r^3 results in a generating set for D_8. For example, $\langle r^2s, r^2, rs \rangle$ allows us to isolate r by multiplying $r^2s \cdot rs = r^2ssr^3 = r$. So $\langle r \rangle \leq \langle r^2s, r^2, rs \rangle$. Thus, $\langle r^2s, r^2, rs \rangle = D_8$.

Example 118.



The Hasse diagram, when combined with Lagrange's Theorem, provides a powerful tool to compute the center, normalizers, and centralizers for a given group. As each of these sets are subgroups of G, they are each vertices on the Hasse diagram. So finding a known subgroup of the center, centralizer, or normalizer, we can narrow down candidates rather quickly. We consider an example.

Example 119. We seek to compute $C_{D_8}(s)$. Recall that $\langle s \rangle \leq C_{D_8}(s)$. Examining the lattice of D_8, our candidates for $C_{D_8}(s)$ are $\langle s, r^2 \rangle$ and D_8. We see that $r^2 \langle s \rangle r^2 = \langle s \rangle$. However, $rs \neq sr$, so $C_{D_8}(s) \neq D_8$.

3.3 Quotient Groups

3.3.1 Introduction to Quotients

Recall in the exposition in the preliminaries that an equivalence relation partitions a set. We refer to this partitioning as a *quotient*, denoted $S/\equiv$, where S is the set and $\equiv$ is the equivalence relation. We pronounce $S/\equiv$ as S modulo $\equiv$. Quotients appear throughout mathematics and theoretical computer science. In automata theory, we study quotient machines and quotient languages, with elegant results such as the Myhill-Nerode Theorem characterizing regular languages using quotients. The Myhill-Nerode theorem also provides an elegant algorithm to compute the quotient and yield a minimum DFA. Section 2.9 of these notes introduces the Myhill-Nerode theorem.

Quotients arise frequently in the study of algebra. In group theory, we study quotients of groups and the conditions upon which the quotient of two groups forms a group itself. In ring theory, we are interested in collapsing the structure to form a field. In fact, we take the ring $\mathbb{R}[x]$ of polynomials with real-valued coefficients and collapse these polynomials modulo $x^2 + 1$ to obtain a field isomorphic to $\mathbb{C}$. Similar constructions produce finite fields of interest in cryptography, such as the Rijndael field used in the AES-Cryptosystem.

In standard algebra texts, the study of group quotients is really restricted to the case when such quotients form a group. Formally, the elements of a group G are partitioned into equivalence classes called *cosets*. In order to partition one group G according to another group H, we use the orbit relation when H acts on G. This yields some interesting combinatorial results, as well as algebraic results in the study of quotient groups.

In particular G/H forms a group when H is a *normal subgroup* of G. This means that H is the kernel of some group homomorphism with G in the domain. These ideas culminate to develop the notion of division, which intuitively speaking comes down to placing an equal number of cookies on each plate.

We begin by computing a couple quotients to illustrate the point. Using quotients of groups, we deduce that $P(n, r) = \frac{n!}{(n-r)!}$ is the correct formula for counting r-letter permutations from an n-letter set; and $\binom{n}{k} = \frac{n!}{r!(n-r)!}$ counts the number of k-element subsets from an n-element set. Recall that S_n is the group of all permutations, with order $n!$. In the mathematical preliminaries section, we considered equivalence classes of permutations in S_n according to whether the "first" r characters were the same. Intuitively, only the last r characters matter in a given permutation. Formally, $P(n, r) = |S_n/S_{n-r}|$. The equivalence classes are formalized by letting S_{n-r} act on S_n by postcomposition. So let $\pi \in S_n$ and $\tau \in S_{n-r}$. Then τ sends $\pi \mapsto \tau \circ \pi$. So the action of S_{n-r} on S_n partitions S_n into orbits each of order $(n-r)!$. So we have $P(n, r) = \frac{n!}{(n-r)!}$ orbits.

Example 120. Consider S_5/S_3. We compute the orbit of (13254). We see:

- $(1)(13254) = (13254)$. Combinatorially, this permutation corresponds to the string 43152.
- $(12)(13254) = (13)(254)$. Combinatorially, this permutation corresponds to the string 34152.
- $(13)(13254) = (3254)$. Combinatorially, this permutation corresponds to the string 13452.
- $(23)(13254) = (1254)$. Combinatorially, this permutation corresponds to the string 41352.
- $(123)(13254) = (254)$. Combinatorially, this permutation corresponds to the string 14352.
- $(132)(13254) = (12543)$. Combinatorially, this permutation corresponds to the string 31452.

So $\mathcal{O}((13254)) = \{(13254), (13)(254), (3254), (1254), (254), (12543)\}$.

By similar argument, we let S_r act on the orbits of S_n/S_{n-r} by postcomposition, which permutes the "last" r digits of the string. We note that the permutations in S_r are labeled using the set $[r]$, while the permutations of S_{n-r} are labeled using the digits $\{r+1, \ldots, n\}$. So the action of S_r on S_n/S_{n-r} does not interfere with the action of S_{n-r} on S_n. Formally, the action of S_r on S_n/S_{n-r} partitions the $\frac{n!}{(n-r)!}$ orbits of S_n/S_{n-r} into equivalence classes each of order $r!$. Intuitively, we are combining orbits of S_n/S_{n-r}. So there are $\frac{n!}{r!(n-r)!} = \binom{n}{r}$ subsets of order r from an n-element set.

Example 121. Recall the example of S_5/S_3 above. We let S_2 act on S_5/S_3. So the following permutations belong to the same orbit:

$$\{(13254), (13)(254), (3254), (1254), (254), (12543), (1324), (13)(24), (324), (124), (24), (1243)\}.$$

So while the string (13254) corresponds to the string 43152, the permutation (1324) corresponds to the string 43125. So the action of S_2 on the orbits of S_5/S_3 permutes the last two digits of a given string.

3.3.2 Normal Subgroups and Quotient Groups

We transition from talking about quotients of sets modulo equivalence relations to developing some intuition about quotient groups, where G/H forms a group. Intuitively, the study of quotient groups is closely tied to the study of homomorphisms. Recall a group homomorphism is a function $\phi : G \to K$ where G and K are groups. Let $H := \ker(\phi)$. Let $a, b \in K$. Intuitively, in a quotient group, we consider $\phi^{-1}(a)$ equivocal to a and $\phi^{-1}(b)$ equivocal to b. That is, $\phi^{-1}(a)$ behaves in G/H just as a behaves in K. That is, the operation of K provides a natural multiplication operation in G/H where multiply orbits by selecting a representative of each orbit, multiplying the representatives and taking the resultant orbit. Using this intuition, we see that $G/H \cong \phi(G)$, which indicates that in the action of H on G, the orbits of this action can be treated equivalently as the range of ϕ. This result is known as the First Isomorphism Theorem, which we will formally prove. Each orbit corresponds to some non-empty preimage of an element in K. It is common in the study of quotients for the orbits or cosets to be referred to as *fibers*. That is, $\phi^{-1}(a)$ is the *fiber above* $a \in K$.

Now consider a group G acting on a set A. In general, the orbits are not invariant when G acts by left multiplication on A vs. right multiplication. In quotient groups, it does not matter if the action is left multiplication or right multiplication. We formalize this as follows.

Proposition 3.14. *Let $\phi : G \to H$ be a group homomorphism with kernel K. Let $X \in G/K$ be the fiber above a; that is, $X = \phi^{-1}(a)$. Then for any $u \in X$, we have $X = \{uk : k \in K\} = \{ku : k \in K\}$.*

Proof. Let $u \in X$. Define $uK = \{uk : k \in K\}$ and $Ku = \{ku : k \in K\}$. We show $uK \subset X$ first. Let $uk \in uK$. Then $\phi(uk) = \phi(u)\phi(k)$ as ϕ is a homomorphism. As $k \in K$, $\phi(k) = 1_H$. So $\phi(u)\phi(k) = \phi(u) = a$. So $uK \subset X$. We now show that $X \subset uK$. Let $g \in X$ and let $k = u^{-1}g$. Observe that $k \in K$, as $\phi(k) = \phi(u^{-1})\phi(g) = a^{-1} \cdot a = 1_H$. So we have $g = uk \in uK$. So $\phi(g) = a$ and $g \in X$. So $X = uK$.

By similar argument, we deduce that $X = Ku$. The details are left to the reader. □

Remark: As the orbit relation is an equivalence relation, each equivalence class can be described by selecting an arbitrary representative. For any $N \leq G$, $gN = \{gn : n \in N\}$ and $Ng = \{ng : n \in N\}$. We refer to gN as the *left coset* and Ng as the *right coset.* If G is an Abelian group, then we write gN as $g + N$; and Ng as $N + g$. By Proposition 3.14, if N is the kernel of some homomorphism, we have that $gN = Ng$.

The first big result in the study of quotient groups is the First Isomorphism Theorem, which we mentioned above. The important result is that $G/\text{ker}(\phi) \cong \phi(G)$ for a group homomorphism $\phi : G \to H$. It is easy to verify that $\phi(G) \leq H$ using the subgroup criterion- an exercise left for the reader. Showing that $G/\text{ker}(\phi)$ forms a group takes some work. Constructing an isomorphism from $G/\text{ker}(\phi)$ to $\phi(G)$ is relatively straight-forward. The desired isomorphism is straight-forward to construct: we map a cost $aG \mapsto \phi(a)$. Note that when we deal with functions on cosets, we must show that the desired function is well-defined. That is, the function is determined for all inputs, and that all members of an equivalence class behave in the expected manner. If a and b belong to the same coset, then it is necessary for a well-defined function f that $f(a) = f(b)$.

We begin by showing $G/\text{ker}(\phi)$ forms a group. We have already discussed the importance of having a well-defined operation. The second part of this proof shows that the desired operation satisfies the group axioms. A good strategy when dealing with quotient groups is to take elements from the quotient group, work in the parent group, apply the homomorphism, then project back into the quotient group. This is precisely what we do below. Furthermore, we note that the desired isomorphism (sending $X = \phi^{-1}(a) \in G/K$ to $a \in \phi(G)$) to prove the First Isomorphism Theorem follows is contained (though not explicitly mentioned) in the proof of this next result.

Theorem 3.16. *Let $\phi : G \to H$ be a group homomorphism with kernel K. Then the operation on G/K sending $aK \cdot bK = (ab)K$ is well-defined.*

Proof. Let $X, Y \in G/K$ and let $Z = XY \in G/K$. Suppose $X = \phi^{-1}(a)$ and $Y = \phi^{-1}(b)$ for some $a, b \in \phi(G)$. Then by the definition of the operation, $Z = \phi^{-1}(ab)$. Let $u \in X, v \in Y$ be representatives of X and Y respectively. It suffices to show $uv \in Z$. We apply the homomorphism ϕ to obtain the that:

$$\begin{aligned}\phi(uv) &= \phi(u)\phi(v)\\ &= ab.\end{aligned}$$

Thus, $uv \in Z$, so $Z = abK$. So the operation is well-defined. □

We now have most of the machinery we need to prove the First Isomorphism Theorem. We want a couple more results first, though, to provide more intuition about the structure of a quotient group. First, we show that the cosets or orbits of an action form a partition of the group. Then we formalize the notion of a normal subgroup. The machinery we build up makes the proof of the First Isomorphism Theorem rather trivial. Remember that our goal is to show that the cosets of G/K behave the same way as the elements of $\phi(G)$, for a group homomorphism $\phi : G \to H$.

Proposition 3.15. *Let G be a group, and let $N \leq G$. The set of left cosets in G/N forms a partition of G. Furthermore, for any $u, v \in G$, $uN = vN$ if and only if $v^{-1}u \in N$. In particular, $uN = vN$ if and only if u, v are representatives of the same coset.*

Proof. As $N \leq G$, $1 \in N$. So for all $g \in G$, $g \cdot 1 \in gN$. It follows that:

$$G = \bigcup_{g \in G} gN.$$

We now show that any two distinct left-cosets are disjoint. Let $uN, vN \in G/N$ be distinct cosets. Suppose to the contrary that $uN \cap vN \neq \emptyset$. Let $x = un = vm \in uN \cap vN$, with $m, n \in N$. As N is a group, $mn^{-1} \in N$. So for any $t \in N$, $ut = vmn^{-1}t = v(mn^{-1}t) \in vN$. So $u \in vN$ and $uN \subset vN$. Interchanging the roles of u and v, we obtain that $vN \subset uN$ and we have $uN = vN$. It follows that $uN = vN$ if and only if $uv^{-1} \in N$ if and only if u, v are representatives of the same coset. □

Remark: In particular, Proposition 3.15 verifies that the action of N on G by right multiplication (the left-coset relation) forms an equivalence relation on G.

We now introduce the notion of a normal subgroup.

Definition 102 (Normal Subgroup). Let G be a group, and let $N \leq G$. We refer to gng^{-1} as the *conjugate* of n by g. The set $gNg^{-1} = \{gng^{-1} : n \in N\}$ is referred to as the *conjugate* of N by g. The element of $g \in G$ is said to *normalize* N if $gNg^{-1} = N$. N is said to be a *normal subgroup* in G if $gNg^{-1} = N$ for all $g \in G$. We denote N to be a normal subgroup of G as $N \trianglelefteq G$.

Intuitively, it is easy to see why a normal subgroup N is the kernel of some homomorphism ϕ. We let G act on N by conjugation. Then for any $g \in G$ and $n \in N$, we consider gng^{-1} and apply ϕ. As $n \in N = \ker(\phi)$, we have $\phi(gng^{-1}) = \phi(g)\phi(n)\phi(g^{-1}) = \phi(g)\phi(g^{-1}) = 1$. We next explore several characterizations of a normal subgroup.

Proposition 3.16. *Let G be a group, and let $N \leq G$. We have the following conditions:*

1. *The operation on the left cosets sending $uN \cdot vN = (uv)N$ is well-defined if and only if $gNg^{-1} \subset N$ for all $g \in G$.*
2. *If the above operation is well-defined, then it makes the set of left-cosets of G/N into a group. The identity of this group is the coset N, and the inverse of gN is $g^{-1}N$.*

Proof. We prove statement (1) first. Suppose the operation is well-defined on G/N. We observe that $g^{-1}N = (nN \cdot g^{-1}N) = (ng^{-1})N$ for any $g \in G$ and $n \in N$. Clearly, $ng^{-1} \in (ng^{-1})N$. As $g^{-1}N = (ng^{-1})N$, we have that $g^{-1}n_1 = ng^{-1}$ for some $n_1 \in N$. Thus, $n_1 = gng^{-1}$. As our choice of g and n were arbitrary, we deduce that $gNg^{-1} \subset N$ for all $g \in G$.

Conversely, suppose $gNg^{-1} \subset N$ for all $g \in G$. Let $u, u_1 \in uN$ and $v, v_1 \in vN$. We write $u_1 = un$ and $v_1 = vm$ for some $m, n \in N$. It suffices to show $u_1v_1 \in (uv)N$. Observe that $u_1v_1 = unvm = u(vv^{-1})nvm$. By associativity, we rewrite $uv(v^{-1}nv)m$. As $gNg^{-1} \subset N$ for all g, we have $v^{-1}nv = n_1$ for some $n_1 \in N$. So $(uv)(v^{-1}nv)m = (uv)(n_1m)$. As $N \leq G$, $n_1m \in N$. So $u_1v_1 \in (uv)N$, completing the proof of (1).

We now prove statement (2). Suppose the operation on G/N sending $uN \cdot vN = (uv)N$is well-defined. We show G/N forms a group. Let $uN, vN, wN \in G/N$. Then $(uN \cdot vN) \cdot wN = uvN \cdot wN = uvwN = uN \cdot (vwN) = uN \cdot (vN \cdot wN)$. So G/N is associative. Let $g \in G$. Observe that $1N = N$; and so, $1N \cdot gN = 1gN = gN$; and $gN \cdot 1N = g1N = gN$. So N is the identity. Now observe that $gN \cdot g^{-1}N = gg^{-1}N = N$. So $(gN)^{-1} = g^{-1}N$. Thus, G/N forms a group. □

We next show that normal subgroups are precisely the kernels of group homomorphisms.

Proposition 3.17. *Let G be a group, and let $N \leq G$. We have $N \trianglelefteq G$ if and only if there exists a group H and group homomorphism $\phi : G \to H$ for which N is the kernel.*

Proof. Suppose first N is the kernel of ϕ. Let G act on N by conjugation. Then $\phi(gNg^{-1}) = \phi(g)\phi(N)\phi(g^{-1}) = \phi(g)\phi(g^{-1}) = 1_H$. So $gNg^{-1} \subset N$. We now show that $gNg^{-1} = N$. Let $g \in G$. The map $\sigma_g : N \to N$ sending $n \mapsto gng^{-1}$ is an injection, as $gn_1g^{-1} = gn_2g^{-1} \implies n_1 = n_2$ by cancellation of the g and g^{-1} terms. Furthermore, the map $gn^{-1}g^{-1}$ is a two-sided inverse of gng^{-1}, so $gNg^{-1} = N$, and we have $N \trianglelefteq G$.

Conversely, suppose $N \trianglelefteq G$. We construct a group homomorphism π for which N is the kernel. By Proposition 3.16, G/N forms a group under the operation sending $uN \cdot vN = (uv)N$. We define the map $\pi : G \to G/N$ sending $g \mapsto gN$. Now let $g, h \in G$. So $\pi(gh) = (gh)N$. By the operation in G/N, $(gh)N = gN \cdot hN = \pi(g)\pi(h)$. So π is a homomorphism. We have $\ker(\pi) = \{g \in G : \pi(g) = 1N\} = \{g \in G : gN = 1N\}$. As $1N = N$, $\ker(\pi) = \{g \in G : gN = N\} = N$. □

We summarize our characterizations of normal subgroups with the next theorem. We have proven most of these equivalences above. The rest are left as exercises for the reader.

Theorem 3.17. *Let G be a group, and let $N \leq G$. The following are equivalent.*

1. *$N \trianglelefteq G$.*
2. *$N_G(N) = G$.*
3. *$gN = Ng$ for all $g \in G$.*
4. *The operation on the left cosets described in Theorem 3.16 forms a group.*
5. *$gNg^{-1} \subset N$ for all $g \in G$.*
6. *N is the kernel of some group homomorphism $\phi : G \to H$.*

We conclude with the First Isomorphism Theorem.

Theorem 3.18 (First Isomorphism Theorem)**.** *Let $\phi : G \to H$ be a group homomorphism. Then $ker(\phi) \trianglelefteq G$ and $G/ker(\phi) \cong \phi(G)$.*

Proof. By Proposition 3.17, we have $\ker(\phi) \trianglelefteq G$. Let $N := \ker(\phi)$. By Proposition 3.16, G/N forms a group. We construct an isomorphism $\pi : G/N \to \phi(G)$ sending $gN \mapsto \phi(g)$. We first show this map is well-defined. Let $g, h \in gN$. As $gN = \phi^{-1}(a)$ for some $a \in H$, we have $\phi(gN) = \phi(g)\phi(N) = a$, as $\phi(g) = a$ and $\phi(N) = 1$. By similar argument, $\phi(hN) = a$. So π is well-defined.

We now show π is an isomorphism. As $G/N = \{\phi^{-1}(a) : a \in \phi(G)\}$, π is surjective. Now suppose $\pi(gN) = \pi(hN) = a$. Then $gN = hN = \phi^{-1}(a)$ and π is injective. Finally, consider $\pi(gN \cdot hN) = \phi(gN \cdot hN)$. As ϕ is a homomorphism, $\phi(gN \cdot hN) = \phi(gN)\phi(hN) = \pi(gN) \cdot \pi(hN)$. So π is a homomorphism. Therefore, π is an isomorphism. □

3.3.3 More on Cosets and Lagrange's Theorem

In this section, we explore some applications of Lagrange's Theorem. In particular, we are able to quickly determine the number of cosets in a quotient, when it is finite. We then examine more subtle results concerning quotients of groups G/H where H is not normal in G. We recall the statement of Lagrange's Theorem below.

Theorem 3.19 (Lagrange's Theorem)**.** *Let G be a finite group, and let $H \leq G$. Then $|H|$ divides $|G|$.*

Recall the proof of Lagrange's Theorem (Theorem 3.9). Intuitively, the proof is analogous to the necklace counting proof of Fermat's Little Theorem. We let H act on G by left multiplication, which partitions the elements of G into orbits of order $|H|$. So $|H|$ divides $|G|$, and we have $\dfrac{|G|}{|H|}$ orbits in G/H. In fact, Lagrange's Theorem implies Fermat's Little Theorem, providing a second proof of Fermat's Little Theorem.

Theorem 3.20 (Fermat's Little Theorem)**.** *Let p be prime and let $a \in [p-1]$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Proof. There are $p - 1$ elements in the multiplicative group $\mathbb{Z}_p^\times$. By Lagrange's Theorem, $|\overline{a}| = |\langle \overline{a} \rangle|$ divides $p - 1$ for every $\overline{a} \in \mathbb{Z}_p^\times$. Let $|\overline{a}| = q$, and $p - 1 = kq$. Then $|\overline{a}|^{p-1} = |\overline{a}^q|^k = \overline{1}^k = \overline{1}$. So $a^{p-1} \equiv 1 \pmod{p}$. □

Remark: More generally, in $\mathbb{Z}_n$ where n is not necessarily prime, $|\mathbb{Z}_n^\times| = \phi(n)$, where ϕ is Euler's totient function. Note that $\phi(n) = |\{a : a \in [n-1], \gcd(a, n) = 1\}|$. The Euler-Fermat Theorem states that $a^{\phi(n)} \equiv 1 \pmod{n}$. So the Euler-Fermat Theorem generalizes and implies Fermat's Little Theorem. The proof is identical to Fermat's Little Theorem, substituting $p - 1$ for $\phi(n)$. Note that if for any prime p, $\phi(p) = p - 1$.

We introduce formal notion for the order of a quotient of groups G/H. Note that we do not assume G/H forms a group.

Definition 103 (Index)**.** Let G be a group, and let $H \leq G$. The *index* of H in G, denoted $[G : H]$ is the number of left cosets in G/H. If G and H are finite, $[G : H] = \dfrac{|G|}{|H|}$. If G is infinite, then $\dfrac{|G|}{|H|}$ does not make sense. However, an infinite group may have a subgroup of finite index. For example, $[\mathbb{Z} : \{0\}] = \infty$ but $[\mathbb{Z} : \langle n \rangle] = n$ for every integer $n > 0$.

We now derive a couple easy consequences of Lagrange's Theorem.

Proposition 3.18. *Let G be a finite group, and let $x \in G$. Then $|x|$ divides $|G|$. Furthermore, $x^{|G|} = 1$ for all $x \in G$.*

Proof. Recall that $|x| = |\langle x \rangle|$. So by Lagrange's Theorem, $|x|$ divides $|G|$. Let $|x| = k$ and $|G| = kq$, for some integer q. Then $x^{|G|} = (|x|^k)^q = 1^q = 1$. $\square$

Proposition 3.19. *If G is a group of prime order p, then $G \cong \mathbb{Z}_p$.*

Proof. Let $H \leq G$. By Lagrange's Theorem, $|H| = 1$ or $|H| = p$. As the identity is the unique element of order 1 and $p > 1$, there exists an element x of order p in G. So $G = \langle x \rangle \cong \mathbb{Z}_p$. $\square$

The converse of Lagrange's Theorem states that if G is a finite group and k divides $|G|$, then G contains a subgroup of order k. In general, the full converse of Lagrange's Theorem does not hold. Consider the following example.

Definition 104 (Alternating Group)**.** Let X be a finite set. Denote $\text{Alt}(X)$ as the group of permutations of X with even order. In particular, $\text{Alt}(n) \leq \text{Sym}(n)$.

Example 122. The elements of $\text{Alt}(4)$ are as follows:

$$\begin{aligned}\text{Alt}(4) = \{&(1), (12)(34), (13)(24), (14)(23), (123), (132),\\ &(143), (134), (124), (142), (243), (234)\}.\end{aligned}$$

Observe that while $|\text{Alt}(4)| = 12$, $\text{Alt}(4)$ has no subgroup of order 6.

We next discuss Cauchy's Theorem, which provides a nice partial converse for Lagrange's Theorem: for every prime divisor p of $|G|$, where G is a finite group, there exists a subgroup of order p in G. Algebra texts introduce Cauchy's Theorem and prove it by induction for Abelian groups. This restricted case is then used to prove the Sylow theorems, which allow us to leverage combinatorial techniques to study the structure of finite groups. The Sylow theorems are then used to prove Cauchy's Theorem in its full generality. We offer an alternative and far more elegant proof of Cauchy's Theorem, which is accredited to James H. McKay. In his proof, McKay uses group actions and combinatorial techniques to prove Cauchy's Theorem, which resembles the necklace-counting (group action) proof of Fermat's Little Theorem we offered in these notes as well as the proof of Lagrange's Theorem.

Theorem 3.21 (Cauchy's Theorem)**.** *Let G be a finite group, and let p be a prime divisor of $|G|$. Then G contains a subgroup of order p.*

Proof. Let:

$$\mathcal{G} = \left\{ (x_1, \ldots, x_p) \in G^p : \prod_{i=1}^{p} x_i = 1 \right\}.$$

The first $p-1$ elements of any tuple in $\mathcal{G}$ may be chosen freely from G. This fixes:

$$x_p = \left(\prod_{i=1}^{p-1} x_i \right)^{-1}.$$

By rule of product, $|\mathcal{G}| = |G|^{p-1}$. As $\prod_{i=1}^{j} x_i$ and $\prod_{i=j+1}^{p} x_i$ are inverses for any $j \in [p]$, $\mathcal{G}$ is closed under cyclic rotations. Let $\mathbb{Z}_p \cong \langle (1, 2, \ldots, p) \rangle$ act on $\mathcal{G}$ by cyclic rotation. Each orbit has order 1 or order p, as p is prime. Let k denote the number of orbits of order 1, and let d denote the number of orbits of order p. We have:

$$|\mathcal{G}| = |G|^{p-1} = k + pd.$$

As p divides $|G|$, p also divides $|G|^{p-1}$. Clearly, p divides pd, so p must divide k. The tuple consisting of all 1's is in $\mathcal{G}$, so $k > 0$. As $k > 0$, $p > 1$, and since p divides k, there must exist a tuple in $\mathcal{G}$ consisting of all x terms for some $x \in G$ with $x \neq 1$. So $x^p = 1$ and we have a subgroup of order p in G. $\square$

Remark: The necklace counting proof of Cauchy's Theorem actually provides that there are at least $p-1$ elements $x \in G$ with $x \neq 1$ satisfying $x^p = 1$.

We conclude by examining another method for constructing subgroups: the concatenation of two subgroups. Recall that we can form subgroups by taking joins, intersections, and under certain conditions quotients of groups. Recall that the concatenation of two sets H and K is the set $HK = \{hk : h \in H, k \in K\}$. When H and K are subgroups of a group G, we evaluate each hk term using the group operation of G and retain the distinct elements. The question arises of when $HK \leq G$. This occurs precisely when $HK = KH$. So it suffices that either $H \trianglelefteq G$ or $K \trianglelefteq G$. However, we can relax this condition. It really suffices that $H \leq N_G(K)$. We begin by determining $|HK|$ in the finite case, using a bijective argument. When HK/K and $H/(H \cap K)$ form groups, the bijection we construct.

Proposition 3.20. *Let G be a group, and let $H, K \leq G$ be finite. Then:* $|HK| = \dfrac{|H| \cdot |K|}{|H \cap K|}$.

Proof. We define $f : H \times K \to HK$ sending $f(h, k) = hk$. Clearly, $|H \times K| = |H| \cdot |K|$. So it suffices to show there exist exactly $H \cap K$ preimages.

Observe that f is surjective, so $f^{-1}(hk) \neq \emptyset$ for all $hk \in HK$. Let $S = \{f^{-1}(hk) : hk \in HK\}$. As f is surjective, $|S| = |HK|$. We define a map $\phi : S \to H/(H \cap K)$ sending $f^{-1}(hk) \mapsto h(H \cap K)$. It suffices to show ϕ is a bijection. Clearly, ϕ is surjective. We now show that ϕ is injective. Suppose $f^{-1}(hk) \neq f^{-1}(h1k_1)$. Then for every $g \in H \cap K$, $h \neq h_1 g$. So $h(H \cap K) \neq h_1(H \cap K)$, and ϕ is injective. So ϕ is a bijection and the result follows. ☐

We offer a second proof of Proposition 3.20 using group actions. This proof relies on the Orbit-Stabilizer Theorem, which we state here. The proof of the Orbit-Stabilizer Theorem is deferred to a later section.

Theorem 3.22 (Orbit-Stabilizer Lemma)**.** *Let G be a group acting on the set A. Then $|Stab(a)| \cdot |\mathcal{O}(a)| = |G|$, where $Stab(a)$ is the stablizer of a and $\mathcal{O}(a)$ is the orbit of a.*

Proof of Proposition 3.20. We let $H \times K$ act on the set $HK \subset G$, where for $(h, k) \in H \times K$ and $x \in HK$:

$$(h, k) \cdot x \mapsto hxk^{-1}.$$

We show that this action is transitive. As $H, K \leq G$, $1 \in HK$. So for $h \in H$ and $k \in K$,

$$(h, k^{-1}) \cdot 1 \mapsto h1k = hk.$$

So $H \times K$ acts transitively on HK. That is, $\mathcal{O}(1) = HK$. We now determine Stab(1). We have that:

$$\begin{aligned} \text{Stab}(1) &= \{(h, k) \in H \times K | h1k^{-1} = 1\} \\ &= \{(h, k) \in H \times K | h1 = k\} \\ &= \{(h, k) \in H \times K | h = k\} \\ &= \{(h, h) \in H \times K\}. \end{aligned}$$

In particular, observe that if $(h, k) \in \text{Stab}(1)$, then $h, k \in H \cap K$. We establish an isomorphism $\varphi : \text{Stab}(1) \to H \cap K$. Let $\varphi((h, h) = h \in H \times K$. This is clearly a surjective map with $\ker(\varphi) = \{1\}$. It remains to show that φ is a group homomorphism. Take $(h, h), (k, k) \in \text{Stab}(1)$. Now:

$$\begin{aligned} \varphi\Big((h, h) \cdot (k, k)\Big) &= \varphi((hk, hk)) \\ &= hk \\ &= \varphi((h, h)) \cdot \varphi((k, k)). \end{aligned}$$

So $\text{Stab}(1) \cong H \times K$. By the Orbit-Stabilizer Theorem, we have that:

$$|H \times K| = |HK| \cdot |H \cap K|.$$

The result follows. ☐

Remark: Let $G = S_3$, $H = \langle (1,2) \rangle$, and $K = \langle (1,3) \rangle$. Then $|H| = |K| = 2$ and $|H \cap K| = 1$. However, by Lagrange's Theorem, $HK \not\leq G$ as $|HK| = 4$, which does not divide $|S_3| = 6$. It follows that $S_3 = \langle (1,2), (1,3) \rangle$. Observe as well that when $HK \leq G$, f is a homomorphism with kernel $H \cap K$. In this case, the First Isomorphism Theorem implies the desired result. The bijective proof presented here provides only the desired combinatorial result.

We now examine conditions in which $HK \leq G$. Observe that:

$$HK = \bigcup_{h \in H} hK.$$

In order for a set of cosets to form a group, it is sufficient and necessary that $hK = Kh$ for all $h \in H$. So it stands to reason that $HK = KH$ needs to hold. Another way to see this is that if $hk \in HK$, we need $(hk)^{-1} = k^{-1}h^{-1} \in HK$ as well. Observe that $k^{-1}h^{-1} \in KH$. So if $HK = KH$, then $k^{-1}h^{-1} \in HK$ and we have closure under inverses. We formalize this result below.

Proposition 3.21. *Let G be a group, and let $H, K \leq G$. We have $HK \leq G$ if and only if $HK = KH$.*

Proof. Suppose first $HK = KH$. We show $HK \leq G$. As $H, K \leq G$, we have $1 \in HK$. So $HK \neq \emptyset$. Now let $a, b \in HK$ where $a = h_1k_1$ and $b = h_2k_2$, $h_1, h_2 \in H$ and $k_1, k_2 \in K$. As $HK = KH$, $k_2h_2 \in HK$. As H and K are groups, $k_2^{-1}h_2^{-1} \in HK$. In order for $ab^{-1} \in HK$, we need $h_1k_1k_2^{-1}h_2^{-1} \in HK$. As $HK = KH$, there exist $k_3 \in K$ and $h_3 \in H$ such that $h_3k_3 = k_1k_2^{-1}h_2$. So $h_1k_1k_2^{-1}h_2^{-1} = h_1h_3k_3 \in HK$ and $HK \leq G$.

Conversely, suppose $HK \leq G$. As H and K are subgroups of HK, we have $KH \subset HK$. In order to show $HK \subset KH$, it suffices to show that for every $hk \in HK$, $(hk)^{-1} \in KH$ (as groups are closed under inverses). Let $hk \in HK$. As HK is a group, $(hk)^{-1} = k^{-1}h^{-1} \in HK$. By definition of KH, we also have that $k^{-1}h^{-1} \in KH$. So $HK \subset KH$, and we conclude that $HK = KH$. $\square$

Remark: Note that $HK = KH$ does not imply that the elements of HK commute. Rather, for every $hk \in HK$, there exists a $k'h' \in KH$ such that $hk = k'h'$. For example, let $H = \langle r \rangle$ and $K = \langle s \rangle$. Then $D_{2n} = HK = KH$, but $sr = rs^{-1}$.

We have a nice corollary to Proposition 3.21.

Corollary 3.21.1. *If H and K are subgroups of G and $H \leq N_G(K)$, then $HK \leq G$. In particular, if $K \trianglelefteq G$, then $HK \leq G$ for all $H \leq G$.*

Proof. By Proposition 3.21, it suffices to show $HK = KH$. Let $h \in H, k \in K$. As $H \leq \text{N}_G(K)$, $hkh^{-1} \in K$. It follows that $hk = (hkh^{-1})h \in KH$. So $HK \subset KH$. By similar argument, $kh = h(h^{-1}kh) \in HK$, which implies $KH \subset HK$. So $HK = KH$. It follows that $HK \leq G$.

Note that if $K \trianglelefteq G$, then $N_G(K) = G$. So any $H \leq G$ satisfies $H \leq N_G(K)$; and thus, $HK \leq G$. $\square$

3.3.4 The Group Isomorphism Theorems

The group isomorphism theorems are elegant results relating a group G to a quotient group G/N. We have already proven the first isomorphism theorem, which states that for a group homomorphism $\phi : G \to H$, $\ker(\phi)$ partitions G into a quotient group isomrophic to $\phi(G)$. The second and fourth isomorphism theorems leverage the poset of subgroups to ascertain the structure of quotients. Finally, the third isomorphism theorem provides us with the "cancellation" of quotients like we would expect with fractions in $\mathbb{Q}$ or $\mathbb{R}$. We have already proven the First Isomorphism Theorem (Theorem 3.18), so we begin with the Second Isomorphism Theorem. The bulk of the machinery to prove the Second Isomorphism Theorem was developed in the last section, so it is a matter of putting the pieces together.

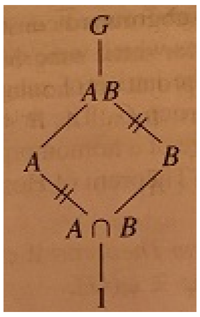
Theorem 3.23 (Second (Diamond) Isomorphism Theorem)**.** *Let G be a group, and let $A, B \leq G$. Suppose $A \leq N_G(B)$. Then $AB \leq G$, $B \trianglelefteq AB$, $A \cap B \trianglelefteq A$, and $AB/B \cong A/(A \cap B)$.*

Proof. By Corollary 3.21.1, we have $AB \leq G$. As $A \leq N_G(B)$ by assumption and $B \leq N_G(B)$, it follows that $AB \leq N_G(B)$. So $B \trianglelefteq AB$. Thus, AB/B is a well-defined quotient group. We define the map $\phi : A \to AB/B$ by sending $\phi(a) = aB$. This map is clearly surjective. We have $\phi(uv) = uvB = uB \cdot vB$ with the last equality by the group operation of AB/B. So $uB \cdot vB = \phi(u)\phi(v)$, and ϕ is a homomorphism. The kernel of this homomorphism is the set:

$$\ker(\phi) = \{a \in A : \phi(a) = B\} = \{a \in A : a \in B\} = A \cap B.$$

So by the First Isomorphism Theorem, we have $A \cap B \trianglelefteq A$, and $A/(A \cap B) \cong \phi(A) = AB/B$. $\square$

The Second Isomorphism Theorem is referred to the as the Diamond Isomorphism Theorem because of the portion of the lattice involved. The marked edges on the lattice indicate the isomorphic quotients.



We now prove the Third Isomorphism Theorem, which considers quotients of quotient groups. Informally, the cancellation property is shown to hold with groups. We also obtain that a quotient preserves normality.

Theorem 3.24 (Third Isomorphism Theorem). *Let G be a group, and let H and K be normal subgroups of G with $H \leq K$. Then $K/H \trianglelefteq G/H$ and $(G/H)/(K/H) \cong G/K$.*

Proof. As H and K are normal in G, we have that G/H and G/K are well-defined quotient groups. We construct a homomorphism $\phi : G/H \to G/K$ sending $\phi(gH) = gK$. We first show ϕ is well-defined. Suppose $g_1H = g_2H$. Then there exists an $h \in H$ such that $g_1 = g_2h$. As $H \leq K$, $h \in K$. So $\phi(g_1H) = \phi(g_2H) = g_1K = g_2K$.

We next argue that ϕ is surjective. Let $gK \in G/K$. We note that $\phi(gH) = gK$. As g may be chosen arbitrarily, ϕ is surjective. Now as ϕ is a projection, it is clearly a homomorphism. It remains to determine $\ker(\phi)$. We have that:

$$\begin{aligned}\ker(\phi) &= \{gH \in G/H : \phi(gH) = K\} \\ &= \{gH \in G/H : g \in K\} \\ &= K/H.\end{aligned}$$

So by the First Isomorphism Theorem, $K/H \trianglelefteq G/H$, and $(G/H)/(K/H) \cong G/K$. $\square$

Remark: The Second and Third Isomorphism Theorems provide nice examples of leveraging the First Isomorphism Theorem. In general, when proving a subgroup H is normal in a parent group G, a good strategy is to constructe a surjective homomorphism from G to some quotient group Q and deduce that H is the kernel of said homomorphism. Projections are often good choices for these types of problems.

We conclude with the Fourth or Lattice Isomorphism Theorem, which relates the lattice of subgroups for the quotient group G/N to the lattice of subgroups of G. Intuitively, taking the quotient G/N preserves the lattice of G, restricting to N as the identity element. This is formalized as follows. The lattice of subgroups of G/N can be constructed from the lattice of subgroups of G, by collapsing the group N to the trivial subgroup and G/N appears at the top of its lattice. In particular, there exists a bijection between the subgroups of G containing N and the subgroups of G/N. We will prove the Fourth Isomorphism Theorem. The general strategy in proving each of these parts is to apply a "lifting" technique, in which we study the quotient structure by taking (under the natural projection homomorphism) the preimages or orbits in G, operating on them, then projecting back down to the quotient. Alternatively, we also study the quotient then lift back to the parent group to study the structure of the original group.

Theorem 3.25 (The Fourth (Lattice) Isomorphism Theorem)**.** *Let G be a group, and let $N \trianglelefteq G$. Then there is a bijection from the set of subgroups A of G containing N onto the set of subgroups $\overline{A} = A/N$ of G/N. In particular, every subgroup of $\overline{G} = G/N$ is of the form A/N for some subgroup A of G containing N (i.e., its preimage in G under the natural projection homomorphism from G to G/N). This bijection has the following properties for all $A, B \leq G$ with $N \leq A$ and $N \leq B$:*

1. *The quotient G/N preserves the subgroups of G. That is, $A \leq B$ if and only if $\overline{A} \leq \overline{B}$.*
2. *The quotient preserves the index. That is, If $A \leq B$, then $[B : A] = [\overline{B} : \overline{A}]$.*
3. *$\overline{\langle A, B\rangle} = \langle \overline{A}, \overline{B}\rangle$.*
4. *$\overline{A \cap B} = \overline{A} \cap \overline{B}$.*
5. *$A \trianglelefteq B$ if and only $\overline{A} \trianglelefteq \overline{B}$.*

Proof. Let $\mathcal{A} = \{A \leq G : N \leq A\}$ and $\mathcal{A}/N = \{A/N \leq G/N\}$. Let $\phi : \mathcal{A} \to \mathcal{A}/N$ be defined sending $A \mapsto A/N$. As ϕ is a projection, ϕ is a homomorphism. Furthermore, ϕ is clearly surjective. We now show that ϕ is injective. Suppose $A_1/N = A_2/N$. As the preimage of a subgroup in a homomorphism is a subgroup of the domain, we have that A_1 and A_2 are subgroups of G containing N, and so $A_1 = A_2$ must hold. So ϕ is injective. We now prove each of the conditions (1)-(5).

1. Suppose first $A \leq B$. For each $a \in A$, we have $\phi(a) = aN$. As $a \in B$, $aN \in \overline{B}$. Since ϕ is a homomorphism, we have $\phi(A) \leq \overline{B}$. Conversely, suppose $\overline{A} \leq \overline{B}$. We apply the subgroup criterion to show $A \leq B$. We clearly have $1N = N \in \overline{A}$, so $1 \in A \cap B$. Now let $\overline{g}, \overline{h} \in \overline{A}$. Then $\overline{h^{-1}} \in \overline{A}$. So $\overline{gh^{-1}} \in \overline{A}$ and $gh^{-1} \in A$. As $\overline{g}, \overline{h} \in \overline{B}$, it follows that $gh^{-1} \in B$ as well. So $A \leq B$.

2. Recall that $[B : A]$ counts the number of left cosets in $\overline{B}$. We map $\psi : B/A \to \overline{B}/\overline{A}$ by projection, sending $bA \mapsto \overline{bA}$. We show ψ is well-defined. Suppose $b_1A = b_2A$. Then $b_2^{-1}b_1A = A$, so $\psi(b_1A) = \psi(b_2A) = \overline{b_1A}$. So ψ is well-defined. Clearly, ψ is surjective. So it suffices to show that ψ is injective. Suppose $\psi(b_1A) = \psi(b_2A)$. Then $\overline{b_1A} = \overline{b_2A}$, which occurs if and only if $\overline{b_2^{-1}b_1A} = \overline{A}$. The preimage is necessarily $b_2^{-1}b_1A = A$ (that is, $b_2A = b_1A$), so ψ is injective.

3. Let $g = \prod_{i=1}^{k} x_i \in \langle A, B\rangle$. Then:
$$\overline{g} = \left(\prod_{i=1}^{k} x_i\right) N = \prod_{i=1}^{k} x_iN.$$
Thus, $\overline{g} \in \langle \overline{A}, \overline{B}\rangle$. Conversely, let $\overline{h} = \prod_{i=1}^{j} \overline{x_i} \in \langle \overline{A}, \overline{B}\rangle$. By construction of G/N, each $x_i = y_in_i$ for some $y_i \in \langle A, B\rangle$ and $n_i \in N$. So:
$$h = \prod_{i=1}^{j} y_i \in \langle A, B\rangle$$
Thus, $\overline{h} \in \overline{\langle A, B\rangle}$ and $\overline{\langle A, B\rangle} = \langle \overline{A}, \overline{B}\rangle$.

4. Let $\overline{h} \in \overline{A \cap B}$, where $\phi(h) = \overline{h}$. So $h \in A \cap B$, which implies that $h \in A$ and $h \in B$. So $\overline{h} \in \overline{A}$ and $\overline{h} \in \overline{B}$. Conversely, let $\overline{g} \in \overline{A} \cap \overline{B}$. So $g = kn$ for some $k \in A \cap B$ and $n \in N$. So $\overline{g} = \overline{k} \in \overline{A \cap B}$. Thus, $\overline{A \cap B} = \overline{A} \cap \overline{B}$.

5. Suppose that $A \trianglelefteq B$. Let $a \in A$ and $b \in B$. We have $\overline{bAb^{-1}} = \overline{b} \cdot \overline{A} \cdot \overline{b^{-1}} = \overline{A}$. So $\overline{A} \trianglelefteq \overline{B}$. Conversely, suppose $\overline{A} \trianglelefteq \overline{B}$. Let $\tau : B \to \overline{B}/\overline{A}$ be given by $\tau(g) = \overline{g}\overline{A}$. We have $\ker(\tau) = \{b \in B : \tau(b) = \overline{A}\}$, which is equivalent to $bN = aN$ for some $a \in A$. As $N \leq A$, there exists $a \in A$ such that $bN = aN$ if and only if $b \in A$. So $\ker(\tau) \subset A$. Conversely, $\tau(A) \subset \ker(\tau)$. So $\ker(\tau) = A$ and $A \trianglelefteq B$.

□

Remark: While the quotient group preserves many properties of its parent, it does not preserve isomorphism. Consider $\mathbb{Q}_8/\langle -1 \rangle \cong D_8/\langle r^2 \rangle \cong \mathbb{V}_4$. However, $Q_8 \not\cong D_8$. We see in the lattices of Q_8 and D_8 below where the sublattice of $\mathbb{V}_4$ is contained.

INCLUDE LATTICES

3.3.5 Alternating Group

In this section, we further discuss the altenrating group of degree n, dentoed $\text{Alt}(n)$. Recall that the $\text{Alt}(n)$ consists of the even permutations of $\text{Sym}(n)$. With the integers, there is a clear notion of even and odd. It is necessary to define an analogous notion for permutations. There are two approaches to formulate the parity of a permutation. The first formulation is to count the number of transpositions in the permutation's decomposition. In order to utilize this latter approach, it must first be shown that a permutation can be written uniquely as the product of disjoint cycles. The permutatoin's parity is then the product of the parity of each cycle in its decomposition.

The second approach is to consider the action of $\text{Sym}(n)$ on the following polynomial:

$$\Delta = \prod_{1 \leq i < j \leq n} (x_i - x_j),$$

which permutes the indices of the variables. That is, for $\sigma \in \text{Sym}(n)$, we have:

$$\sigma(\Delta) = \prod_{1 \leq i < j \leq} (x_{\sigma(i)} - x_{\sigma(j)}).$$

Example 123. Suppose $n = 4$. Then:

$$\Delta = (x_1 - x_2)(x_1 - x_3)(x_1 - x_4)(x_2 - x_3)(x_2 - x_4)(x_3 - x_4).$$

If $\sigma = (1, 2, 3, 4)$, then:

$$\sigma(\Delta) = (x_2 - x_3)(x_2 - x_4)(x_2 - x_1)(x_3 - x_4)(x_3 - x_1)(x_3 - x_4).$$

Here, we wrote the factors of $\sigma(\Delta)$ in the same order as σ. Observe that Δ has the factor $(x_i - x_j)$ for every $1 \leq i < j \leq n$. As σ is a bijection, $\sigma(\Delta)$ has either the factor $(x_i - x_j)$ or $(x_j - x_i)$. Observe that $(x_i - x_j) = -(x_j - x_i)$. It follows that $\sigma(\Delta) = \pm\Delta$ for every $\sigma \in S_n$. We define the parity homomorphism $\epsilon : \text{Sym}(n) \to \{\pm 1\}$ as follows:

$$\epsilon(\sigma) = \begin{cases} 1 & : \sigma(\Delta) = \Delta, \\ -1 & : \sigma(\Delta) = -\Delta. \end{cases}$$

We use ϵ to define the sign or parity of a permutation.

Definition 105 (Sign of a Permutation)**.** The *sign* of the permutation σ is $\epsilon(\sigma)$. If $\epsilon(\sigma) = 1$, then σ is said to be *even*. Otherwise, σ is said to be *odd*.

In particular, ϵ is a homomorphism, which we easily verify below.

Proposition 3.22. *The function ϵ defined above is a homomorphism, where $\{\pm 1\} \cong \mathbb{Z}_2$ using the operation of multiplication for $\{\pm 1\}$.*

Proof. Let $\sigma, \tau \in \text{Sym}(n)$. By definition:

$$(\tau\sigma)(\Delta) = \prod_{1 \leq i < j \leq n} (x_{\tau\sigma(i)} - x_{\tau\sigma(j)}) = \epsilon(\tau\sigma)\Delta$$

We now see that:

$$\begin{aligned}(\tau\sigma)(\Delta) &= \tau(\sigma(\Delta)) \\ &= \tau(\epsilon(\sigma)\Delta) \\ &= \epsilon(\sigma)\tau(\Delta) \\ &= \epsilon(\sigma)\epsilon(\tau)\Delta.\end{aligned}$$

where the first equality follows from the associativity of the group action. So ϵ is a homomorphism. □

In particular, it follows that transpositions are odd permutations and ϵ is a surjective homomorphism. We now define the Alternating group of degree n as follows.

Definition 106 (Alternating Group)**.** Let $n \in \mathbb{N}$, and consider $\epsilon : \text{Sym}(n) \to \{\pm 1\}$, as defined above. The *Alternating group of degree n*, denoted $\text{Alt}(n) := \ker(\epsilon)$.

By Lagrange's Theorem, $|A_n|$ divides $|S_n|$. Furthermore, as ϵ is a homomorphism onto $\{\pm 1\}$, we see that $[\text{Sym}(n) : \text{Alt}(n)] = 2$. So $|\text{Alt}(n)| = \frac{1}{2}|\text{Sym}(n)|$. That is, there are as many even permutations as odd permutations. We also see the map $\psi : \text{Sym}(n) \to \text{Sym}(n)$ sending $\sigma \mapsto \sigma \cdot (12)$ is a bijection. As $\epsilon(12) = -1$, ψ maps even permutations to odd permutations, and odd permutations to even permutations. This provides a bijective argument that there are just as many even permutations as odd permutations.

It is also easy to see why the Alternating group is the kernel of the parity homomorphism. Recall from homework that every permutation can be written as the product of transpositions. We first recognize that $\epsilon(\sigma) = \epsilon(\sigma^{-1})$. Let $\sigma = \prod_{i=1}^{k} s_i$, where each s_i is a transposition. Then $\sigma^{-1} = \prod_{i=1}^{k} s_{k-i+1}$. Intuitively, each transposition in the decomposition of σ needs to be cancelled to obtain the identity. Let S_n act on itself by conjugation. We consider $\epsilon(\sigma\tau\sigma^{-1}) = \epsilon(\sigma)\epsilon(\tau)\epsilon(\sigma^{-1})$. As $\epsilon(\sigma) = \epsilon(\sigma^{-1})$, we have $\epsilon(\sigma)\epsilon(\tau)\epsilon(\sigma^{-1}) = \epsilon(\tau) = 1$ if and only if τ is even and for all $\sigma \in \text{Sym}(n)$.

We now seek to define the Alternating group in terms of the cycle decomposition. Recall that every permutation has a cycle decomposition. In Section 3.1.3, an algorithm was presented to compute the cycle decomposition of a permutation. This algorithm is formally justified using a group action. That is, we prove the cycle decomposition from this algorithm is unique. In order to prove this result, we need a result known as the Orbit-Stablizer Lemma which is also known as Burnside's Lemma. The Orbit-Stabilizer Lemma is a powerful tool in algebraic combinatorics, which is the foundation for Polyá Enumeration Theory.

Theorem 3.26 (Orbit-Stabilizer Lemma)**.** *Let G be a group acting on the set A. Then $|Stab(a)| \cdot |\mathcal{O}(a)| = |G|$, where $Stab(a)$ is the stablizer of a and $\mathcal{O}(a)$ is the orbit of a.*

Proof. Recall that the orbits of a group action partition A (formally the equivalence relation on A is defined as $b \equiv a$ if and only if $b = g \cdot a$ for some $g \in G$). Fix $a \in A$. Recall that $\mathcal{O}(a) = \{g \cdot a : g \in G\}$. So we map $\varphi : \mathcal{O}(a) \to G/\text{Stab}(a)$ by sending $g \cdot a \mapsto g\text{Stab}(a)$. This map is clearly surjective. Now suppose $g \cdot \text{Stab}(a) = h \cdot \text{Stab}(a)$. Recall that $g \cdot a = h \cdot a$ if and only if $h^{-1}g \cdot a = a$, which is equivalent to $h^{-1}g \in \text{Stab}(a)$. So $g\text{Stab}(a) = h\text{Stab}(a)$ if and only if $h^{-1}g\text{Stab}(a) = \text{Stab}(a)$. So this map is injective. It follows that $|\text{Stab}(a)| \cdot |\mathcal{O}(a)| = |G|$, as desired. □

We now prove the existence and uniqueness of the cycle decomposition of a permutation.

Theorem 3.27. *Every permutation $\sigma \in Sym)n)$ can be written uniquely as the product of disjoint cycles.*

Proof. Let $\sigma \in \text{Sym}(n)$, and let $G = \langle \sigma \rangle$ act on $[n]$. Let $x \in [n]$ and consider $\mathcal{O}(x)$. By the Orbit-Stabilizer Lemma, the map $\sigma^i x \mapsto \sigma^i \text{Stab}(x)$ is a bijection. As G is cyclic, $\text{Stab}(x) \trianglelefteq G$, so $G/\text{Stab}(x)$ is a well-defined quotient group. In particular, $G/\text{Stab}(x)$ is cyclic, and $d := |G/\text{Stab}(x)|$ is the least positive integer such that $\sigma^d \in \text{Stab}(x)$. By the Orbit-Stabilizer Lemma $[G : \text{Stab}(x)] = |\mathcal{O}(x)| = d$. It follows that the distinct left-cosets of $G/\text{Stab}(x)$ are $\text{Stab}(x), \sigma\text{Stab}(x), \ldots, \sigma^{d-1}\text{Stab}(x)$, and $\mathcal{O}(x) = \{x, \sigma(x), \sigma^2(x), \ldots, \sigma^{d-1}(x)\}$. We iterate on this argument for each orbit to obtain a cycle decomposition for σ. The uniqueness of the cycle decomposition for σ follows from our selection of σ and the fact that a permutation is a bijection. □

Remark: Theorem 3.27 provides an algorithm for computing the cycle decomposition of a given permutation. We can further decompose each disjoint cycle of σ into a product of transpositions, giving us a factorization of σ in terms of transpositions. So by the Well-Ordering Principle, there exists a minimum number of transpositions whose product forms σ. We then say a permutation σ is even (odd) if its minimum factorization in terms of transpositions consists of an even (odd) number of 2-cycles. The Alternating group of degree n, $\text{Alt}(n)$, can then be defined as the group of even permutations.

3.3.6 Algebraic Graph Theory- Graph Homomorphisms

In this section, we explore some basic results on graph homomorphisms. Recall that a graph homomorphism.

Definition 107 (Graph Homomorphism)**.** Let G and H be graphs. A *graph homomorphism* is a function $\varphi : V(G) \to V(H)$ such that if $ij \in E(G)$, then $\varphi(i)\varphi(j) \in E(H)$. That is, a graph homomorphism preserves the adjacency relation from G into H.

A well-known class of graph homomorphism is the class of graph colorings. A graph G is r-colorable if there exists a homomorphism $\varphi : V(G) \to V(K_r)$. That is, the vertices of K_r are the r-colors of G. For $r \geq 3$, it is an `NP`-Complete problem to decide if a graph G is r-colorable. This is equivalent to deciding if there exists a homomorphism $\phi : V(G) \to K_r$. So it is also `NP`-Complete to decide if there even exists a homomorphism between graphs G and H.

The theory of graph homomorphisms has a similar flavor to the study of group homomorphisms. In a group homomorphism, the operation is preserved in the image. So a product in the domain translates to a product in the codomain. Graph homomorphisms similarly map walks in the domain to walks in the image. Much of what we know about group homomorphisms holds true for graph homomorphisms. One example of this deals with the composition of graph homomorphisms. Let $g : V(H) \to V(K)$ with $h : V(G) \to V(H)$ be graph homomorphisms. Then $g \circ h : V(G) \to V(K)$ is itself a graph homomorphism.

We now define the binary relation $\to$ on the set of finite graphs, where $X \to Y$ if there exists a homomorphism $\phi : V(X) \to V(Y)$. Clearly, $\to$ is reflexive, as $X \to X$ by the identity map. As the composition of two graph homomorphisms is a graph homomorphism, we have that $\to$ is transitive. However, $\to$ fails to be a partial order. Let X be a bipartite graph, and $Y := K_2$. Then there exists a homomorphism from X to Y, mapping one part of X to $v_1 \in Y$ and the other part of X to $v_2 \in Y$. Similarly, if there is an edge in X, there exists a homomorphism from Y to X mapping Y as some edge in X. However, any case when $|X| > |2|$ results in $X \not\cong Y$. We need surjectivity of the homomorphisms from $X \to Y$ and $Y \to X$ to deduce that $X \cong Y$. If $X \to Y$ and $Y \to X$, we say that X and Y are *homomorphically equivalent*.

Much in the same way that we study quotient groups, we study quotient graphs. Let $f : V(X) \to V(Y)$ be a graph homomorphism. The preimages $f^{-1}(y)$ for each $y \in Y$ are the *fibers*, which partition the graph X. We refer to the partition as the *kernel*. In group theory, we refer to the kernel as the preimage of the identity in the codomain. The kernel then acts on the domain, partitioning it into a quotient group isomorphic to the image. In the graph theoretic setting, there is no identity element as there is no operation. So in a more general setting, we view the kernel as an equivalence relation π on the vertices of X. We construct a *quotient graph* X/π as follows. The fibers of π are the vertices of X/π. Then vertices u, v in X/π are adjacent if there exist representatives $a \in f^{-1}(u), b \in f^{-1}(v)$ such that $ab \in E(X)$. Note that if X has loops, it may be the case that $u = v$. There exists a natural homomorphism $\phi : V(X) \to V(X/\pi)$ sending $v \mapsto f(v)$.

While deciding if there exists a homomorphism $\phi : V(X) \to V(Y)$ is `NP`-Complete, we can leverage a couple invariants to make our life easier. First, if the graph Y is r-colorable and there exists a homomorphism from X to Y, then $\chi(X) \leq \chi(y) = r$. This follows from the fact that for graph homomoprhisms $g : V(Y) \to K_r$ and $f : V(X) \to V(Y)$, $g \circ f : V(X) \to K_r$ is a graph homomorphism.

We prove a second invariant based on the *odd girth*, or length of the shortest odd cycle in a graph.

Proposition 3.23. *Let X and Y be graphs, and let $\ell(X)$ be the odd girth in X. If there exists a graph homomorphism $f : V(X) \to V(Y)$, then $\ell(Y) \leq \ell(X)$.*

Proof. Let $v_0, v_1, \ldots, v_{\ell-1}, v_0$ be the sequence of vertices in X that form a cycle of length ℓ, with $v_0 = v_\ell$. Applying f, we obtain $f(v_i)f(v_{i+1}) \in E(Y)$ for each $i \in \{0, \ldots, \ell - 1\}$ with the indices taken modulo ℓ. So

$f(v_0)f(v_1)\dots f(v_{\ell-1})f(v_0)$ is a closed walk of odd length. By Lemma 1.1, $f(v_0)f(v_1)\dots f(v_{\ell-1})f(v_0)$ contains an odd cycle, which implies $\ell(Y) \leq \ell(X)$. □

We now introduce the notion of a *core*. Formally, we have the following.

Definition 108 (Core)**.** A graph X is a *core* if every homomorphism $\varphi : V(X) \to V(X)$ is a bijection. That is, every homomorphism from a core to itself is an automorphism.

Example 124. The simplest class of cores is the set of complete graphs. Odd cycles are also cores. We verify that odd cycles are cores below.

Proposition 3.24. *Let $n \in \mathbb{Z}^+$ and let $X := C_{2n+1}$ be an odd cycle. Let $f : V(X) \to V(X)$ be a homomorphism. Then f is a bijection.*

Proof. Suppose to the contrary that f is not a bijection. Then there exist vertices, which we call v_1, v_j, and v_k such that $1 < j < 2n+1$ and $f(v_1) = f(v_j) = v_k$. Let $P := v_1v_2 \cdots v_j$ be a path in C_{2n+1}. As $f(v_1) = f(v_j)$, $f(P)$ is a cycle of length less than $2n+1$. However, C_{2n+1} contains no smaller cycles, a contradiction. □

Cores provide a useful invariant to decide if there exists a homomorphism from $X \to Y$. We first show that X and Y being isomorphic cores is equivalent to X and Y being homomorphically equivalent. Next, we show that every graph has a core, and a graph's core is unique up to isomorphism. This implies that the relation $\to$ is a partial order on the class of cores.

Definition 109 (Core of a Graph)**.** Let X be a graph. The subgraph Y of X is said to be a *core* of X if Y is a core and there exists a homomorphism from X to Y. We denote the core of X as $X^\bullet$.

We introduce another example of a core, which relates to coloring.

Definition 110 (χ-Critical Graph)**.** A graph X is χ-critical if any proper subgraph of X has chromatic number less than $\chi(X)$.

Remark: In particular, a χ-critical graph cannot have a homomorphism to any of its proper subgraphs. So a χ-critical graph is a core (and therefore, its own core).

Lemma 3.6. *Let X and Y be finite cores. Then X and Y are homomorphically equivalent if and only if they are isomorphic.*

Proof. If $X \cong Y$, then the isomorphisms from X to Y and Y to X are homomorphisms and we are done. Conversely, let $f : V(X) \to V(Y)$ and $g : V(Y) \to V(X)$ be homomorphisms. Then $g \circ f : V(Y) \to V(Y)$ and $f \circ g : V(X) \to V(X)$ are homomorphisms. As X and Y are cores, $g \circ f$ and $f \circ g$ are automorphisms. In particular, f and g are necessarily surjective. As f and g are surjective homomorphisms and X and Y are finite, f and g are necessarily injective. Therefore, f and g are isomorphisms. So $X \cong Y$, as desired. □

We introduce the definition of a retract and induced subgraph before proving the next lemma.

Definition 111 (Retract)**.** Let X be a graph. The subgraph Y of X is said to be a *retract* if there exists a homomorphism $f : X \to Y$ such that the restriction of f to Y is the identity map. We refer to f as a *retraction.*

Definition 112 (Induced Subgraph)**.** Let X be a graph, and let Y be a subgraph of X. The graph Y is said to be an *induced subgraph* of X if $E(Y) = \{ij \in E(X) : i, j \in V(Y)\}$.

Lemma 3.7. *Every graph X has a core, which is an induced subgraph and is unique up to isomorphism.*

Proof. As X is finite and the identity map is a homomorphism, there is a finite and non-empty set of subgraphs Y of X such that $X \to Y$. So there exists a minimal element H with respect to inclusion. Let $f : X \to H$ be a homomorphism. As H is minimal, f restricted to H is an automorphism ϕ of H. Composing f with ϕ^{-1} yields the identity map on H. So H is a retract, and therefore a core. We note that as H is a retract, H is an induced subgraph of X.

Now suppose H_1, H_2 are cores of X. Let $f_i : V(X) \to V(H_i)$ be a homomorphism. Then for each $i \in [2]$, f_i restricted to H_{-i} is a homomorphism from H_i to H_{-i} (where $-i \in [2] - \{i\}$). So by Lemma 3.6, $H_1 \cong H_2$. □

We are now able to characterize homomorphic equivalence in terms of cores.

Theorem 3.28. *Two graphs X and Y are homomorphically equivalent if and only if their cores are isomorphic.*

Proof. Suppose first X and Y are homomorphically equivalent. We note that as $X^\bullet$ is the core of X, the identity map $\text{id} : V(X^\bullet) \to V(X)$ is a graph homomorphism. As $Y^\bullet$ is the core of Y, we have that $Y \to X$. So we have a sequence of homomorphisms:

$$X^\bullet \to X \to Y \to Y^\bullet.$$

These homomorphisms compose to form a homomorphism from $X^\bullet$ to $Y^\bullet$. By similar argument, there exists a homomorphism from $Y^\bullet$ to $X^\bullet$. Lemma 3.6 implies that $X^\bullet \cong Y^\bullet$.

Conversely, suppose $X^\bullet \cong Y^\bullet$. Then we have the following sequences of homomorphisms:

$$X \to X^\bullet \to Y^\bullet \to Y, \text{ and}$$
$$Y \to Y^\bullet \to X^\bullet \to X.$$

Each of these sequences composes to form a homomorphism from X to Y and from Y to X respectively, so X and Y are homomorphically equivalent. □

We now discuss basic results related to cores of vertex-transitive graphs. These results are quite elegant, powerful, and simple. Furthermore, they provide nice analogs to group theoretic results such as Lagrange's Theorem and group actions. We begin by showing that the core of a vertex transitive graph is also vertex transitive.

Theorem 3.29. *Let X be a vertex transitive graph. Then the core of X, $X^\bullet$, is also vertex transitive.*

Proof. Let $x, y \in V(X^\bullet)$ be distinct. Then there exists $\phi \in \text{Aut}(X)$ such that $\phi(x) = y$. Let $f : X \to X^\bullet$ be a retraction. The composition $f \circ \phi : X \to X^\bullet$ forms a homomorphism whose restriction to $X^\bullet$ is an automorphism of $X^\bullet$ mapping $x \mapsto y$. So $X^\bullet$ is vertex transitive. □

Our next theorem provides an analog of Lagrange's Theorem in the case of cores of vertex transitive graphs. Recall the proof of Lagrange's Theorem used group actions. In this next result, we use the core and the homomorphism to partition the parent graph into parts of equal cardinality, which is analogous to a group action.

Theorem 3.30. *Let X be a vertex transitive graph with the core $X^\bullet$. Then $|X^\bullet|$ divides $|X|$.*

Proof. Let $\phi : X \to X^\bullet$ be a surjective homomorphism, and let $\gamma : X^\bullet \to X$ be a homomorphism. It suffices to show each fiber of ϕ has the same order. Let $u \in X^\bullet$. Define the set S as follows:

$$S = \{(v, \psi) : v \in V(X^\bullet), \psi \in \text{Aut}(X), \text{ and } (\phi \circ \psi \circ \gamma)(v) = u\}.$$

We count S in two ways. As ϕ, ψ, and γ are all homomorphisms and $X^\bullet$ is a core, $\phi \circ \psi \circ \gamma \in \text{Aut}(X^\bullet)$. As ϕ and γ are fixed, there exists a unique v dependent only on ψ such that $(\phi \circ \psi \circ \gamma)(v) = u$. So $|S| = |\text{Aut}(X)|$.

We now count S in a second way. As $(\phi \circ \psi \circ \gamma)(v) = u$, we have that $(\psi \circ \gamma)(v) \in \phi^{-1}(u)$. We select $v \in V(X^\bullet)$, $x \in \phi^{-1}(u)$, and an automorphism ψ mapping $\gamma(v) \mapsto x$. There are $|X^\bullet|$ ways to select v and $|\phi^{-1}(u)|$ ways to select x. These selections are independent; so by rule of product, we multiply $|X^\bullet| \cdot |\phi^{-1}(u)|$. Now the set of automorphisms mapping $\gamma(v) \to x$ is a left-coset of $\text{Stab}(\gamma(v))$, which has cardinality $|\text{Stab}(\gamma(v))|$. As X is vertex transitive, the orbit of v under the action of $\text{Aut}(X)$ is $V(X)$. So by the Orbit-Stabilizer Lemma, $|\text{Stab}(\gamma(v))| = |\text{Aut}(X)|/|X|$. By rule of product, $|S| = |X^\bullet| \cdot |\phi^{-1}(u)| \cdot |\text{Aut}(X)|/|X|$. As $|\text{Aut}(X)| = |S|$, we deduce that $|\phi^{-1}(u)| = |X|/|X^\bullet|$. So $|X^\bullet|$ divides $|X|$ and we are done. □

Theorem 3.30 provides a couple nice corollaries. The first is an analog of group theory, which states that a group of prime order p is isomorphic to $\mathbb{Z}_p$. The second corollary provides conditions to deduce when a graph is triangle free.

Corollary 3.24.1. *If X is a connected vertex transitive graph of prime order p, then X is a core.*

Proof. As X is connected and has prime order, $X \not\to K_1$. So $|X^\bullet| = |X|$ and $X^\bullet \to X$. So $X \cong X^\bullet$. □

Corollary 3.24.2. *Let X be a vertex transitive graph of order n, with $\chi(X) = 3$. If n is not a multiple of 3, then X is triangle-free.*

Proof. As $\chi(X) = 3$, there exists a homomorphism from X to K_3. If K_3 was the core of X, then K_3 would be contained in X. By Theorem 3.30, 3 would divide n, a contradiction. $\square$

We next introduce the notion of graph product, which is analogous to the direct product of groups. In group theory, the direct product of $G \times H$ is the set of ordered pairs $\{(g, h) : g \in G, h \in H\}$ with the operation preserved componentwise. That is, $(a, b)(c, d) = (ac, bd)$ where ac is evaluated in G and bd is evaluated in H. The graph product is based on this idea, preserving the adjacency relation component wise.

Definition 113 (Graph Product)**.** Let X and Y be graphs. Then the product $X \times Y$ is the grah with the vertex set $\{(x, y) : x \in V(X), y \in V(Y)\}$ and two vertices $(a, b), (c, d)$ in $X \times Y$ are adjacent if and only if $ac \in E(X)$ and $bd \in E(Y)$.

Remark: Note that the graph product is **not** a Cartesian product. In algebraic graph theory, the Cartesian product of two graphs is denoted as $X \square Y$ and is defined differently than the graph product above.

In a graph product, we have $X \times Y \cong Y \times X$, with the isomorphism sending $(x, y) \mapsto (y, x)$. So factors in a product graph may be reordered in the product. However, a graph may have multiple factorizations. We see that:

$$K_2 \times 2K_3 \cong 2C_6 \cong K_2 \times C_6.$$

So $X \times Y \cong X \times Z$ does not imply that $Y \cong Z$. We also note that for a fixed $x \in V(X)$, the vertices of $X \times Y$ of the form $\{(x, y) : y \in Y\}$ form an independent set. So $X \times K_1$ is the empty graph of order $|X|$, rather than X.

We have already seen in the study of quotient groups that the natural projection homomorphism is quite useful. The natural projection homomorphism is also a common tool in studying direct products of groups, and it comes up frequently in the study of graph homomorphisms. Formally, if we have the product graph $X \times Y$, the projection map: $p_X : (x, y) \mapsto x$ is a homomorphism from $X \times Y \to X$. There is similarly a projection $p_Y : X \times Y \to Y$. We use the projection map to count homomorphisms from a graph Z to a product graph $X \times Y$. We denote the set of homomorphisms from a graph G to a graph H as $\text{Hom}(G, H)$. Our next theorem provides a bijection from:

$$\text{Hom}(Z, X \times Y) \to \text{Hom}(Z, X) \times \text{Hom}(Z, Y).$$

Theorem 3.31. *Let X, Y and Z be graphs. Let $f : Z \to X$ and $g : Z \to Y$ be homomorphisms. Then there exists a unique homomorphism $\phi : Z \to X \times Y$ such that $f = p_X \circ \phi$ and $g = p_Y \circ \phi$.*

Proof. The map $\phi : z \mapsto (f(z), g(z))$ is clearly a homomorphism from Z to $X \times Y$. Furthermore, we have $f = p_X \circ \phi$ and $g = p_Y \circ \phi$. The homomorphism ϕ is uniquely determined by our selections of f and g. So the map $\phi \mapsto (f, g)$ is a bijection. $\square$

Corollary 3.24.3. *For any graphs X, Y, and Z, we have:*

$$|Hom(Z, X \times Y)| = |Hom(Z, X)| \cdot |Hom(Z, Y)|$$

3.3.7 Algebraic Combinatorics- The Determinant

TODO

3.4 Group Actions

3.4.1 Conjugacy

In this section, we explore results related to the action of conjugation. Recall that G acts on the set A by conjugation, with $g \in G$ sending $g : a \mapsto gag^{-1}$. We focus on the case when G acts on itself by conjugation.

Definition 114 (Conjugacy Classes)**.** We say that two elements $a, b \in G$ are *conjugate* if there exists a $g \in G$ such that $b = gag^{-1}$. That is, a and b are conjugate if they belong to the same orbit when G acts on itself by conjugation. Similarly, we say that two subsets of G, S and T, are conjugate if $T = gSg^{-1}$ for some $g \in G$. We refer to these orbits as *conjugacy classes*.

Example 125. If G is Abelian, the action of G on itself by conjugation is the trivial action because $gag^{-1} = gg^{-1}a = a$.

Example 126. When Sym(3) acts on itself by conjugation, the conjugacy classes are $\{1\}$, $\{(1,2),(1,3),(2,3)\}$, $\{(1,2,3),(1,3,2)\}$.

Remark: In particular, if $|G| > 1$; then under the action of conjugation, G does not act transitively on itself. We see that $\{1\}$ is always a conjugacy class, so there are at least two orbits under this action.

We now use the Orbit-Stabilizer Lemma to compute the order of each conjugacy class.

Proposition 3.25. *Let G be a group, and let $S \subset G$. The number of conjugates of S is the index of the normalizer in G, $[G : N_G(S)]$.*

Proof. We note that $\text{Stab}(S) = \{g \in G : gSg^{-1} = S\} = N_G(S)$. The conjugates of S lie in the orbit $\mathcal{O}(S) = \{gSg^{-1} : g \in G\}$. The result follows from the Orbit-Stabilizer Lemma. □

As the orbits partition the group, orders of the conjugacy classes add up to $|G|$. This observation provides us with the Class Equation, which is a powerful tool in studying the orbits in the action of conjugation.

Theorem 3.32 (The Class Equation). *Let G be a finite group, and let $g_1, \ldots, g_r$ be represnetatives of the distinct conjugacy classes in G that are not contained in $Z(G)$. Then:*

$$|G| = |Z(G)| + \sum_{i=1}^{r} [G : C_G(g_i)].$$

Proof. We note that for a single $g_i \in G$, $N_G(g_i) = C_G(g_i)$. So $\text{Stab}(g_i) = C_G(g_i)$. We note that for an element $x \in Z(G)$, $gxg^{-1} = x$ for all $g \in G$. So the conjugacy class of x contains only x. Thus, the conjugacy classes of G are:

$$\{1\}, \{z_2\}, \ldots, \{z_m\}, \mathcal{K}_1, \ldots, \mathcal{K}_r,$$

where $z_2, \ldots, z_m \in Z(G)$ and $g_i \in \mathcal{K}_i$ for each $i \in [r]$. As the conjugacy classes partition G and $|\mathcal{K}_i| = [G : C_G(g_i)]$ by Proposition 3.25, we obtain:

$$\begin{aligned} |G| &= \sum_{i=1}^{m} 1 + \sum_{i=1}^{r} |\mathcal{K}_i| \\ &= |Z(G)| + \sum_{i=1}^{r} [G : C_G(g_i)]. \end{aligned}$$

□

We consider some examples to demonstrate the power of the Class Equation.

Example 127. Let $G = D_8$. We use the class equation to deduce the conjugacy classes of D_8. We first note $Z(D_8) = \{1, r^2\}$, which yields the conjugacy classes $\{1\}$ and $\{r^2\}$. It will next be shown that for each $x \not\in Z(D_8)$, $|C_G(x)| = 4$. As $C_G(x) = \text{Stab}(x)$ under the action of conjugation, the Orbit-Stabilizer Lemma gives us that the remaining conjugacy classes have order 2.

Recall that the three subgroups of order 4 in D_8 are $\langle r \rangle, \langle s, r^2 \rangle$, and $\langle sr, r^2 \rangle$. Each of these subgroups is Abelian. For any $x \not\in Z(D_8)$, $\langle x \rangle \leq C_{D_8}(x)$ and $Z(D_8) \leq C_{D_8}(x)$. So by Lagrange's Theorem, $|C_{D_8}(x)| \geq 4$. As $x \not\in Z(D_8)$, some element of G does not commute with x. So $|C_{D_8}(x)| \leq 7$. So by Lagrange's Theorem, $|C_{D_8}(x)| = 4$. So D_8 has three conjugacy classes of order 2, and two conjugacy classes of order 1, which are listed below:

$$\{1\}, \{r^2\}, \{r, r^3\}, \{s, sr^2\}, \{sr, sr^3\}$$

We next use the class equation to prove that every group of prime power order has a non-trivial center.

Theorem 3.33. *Let P be a group of order p^α for a prime p and $\alpha \geq 1$. Then $Z(P) \neq 1$.*

Proof. If P is Abelian, then $Z(P) = P$. So suppose P is not Abelian. Then there exists at least one element $g \in P$ such that $g \notin Z(P)$. Suppose the distinct conjugacy classes of P are:

$$\{1\}, \{z_2\}, \ldots, \{z_m\}, \mathcal{K}_1, \ldots, \mathcal{K}_r.$$

Let $g_1, \ldots, g_r$ be distinct representatives of $\mathcal{K}_1, \ldots, \mathcal{K}_r$ respectively. As no conjugacy class is equal to P, p divides each $|\mathcal{K}_i| = [P : C_P(g_i)]$. By the class equation, we have:

$$|P| = |Z(P)| + \sum_{i=1}^{r} |\mathcal{K}_i|.$$

As p divides $|P|$ and p divides each $|\mathcal{K}_i|$, p must divide $|Z(P)|$. So $Z(P) \neq 1$. □

Theorem 3.33 provides a nice corollary, allowing us to easily classify groups of order p^2 where p is prime.

Corollary 3.25.1. *Let P be a group of order p^2. Then $P \cong \mathbb{Z}_{p^2}$ or $P \cong \mathbb{Z}_p \times \mathbb{Z}_p$.*

Proof. By Theorem 3.33, $Z(P) \neq 1$. If P has an element of order p^2, then $P \cong \mathbb{Z}_{p^2}$ and we are done. So suppose instead that all every non-identity element has order p. Let $x \in P$ have order p, and let $y \in P \setminus \langle x \rangle$ have order p. Observe that $\langle x \rangle \cap \langle y \rangle = 1$, so $p^2 = |\langle x, y \rangle| > |\langle x \rangle| = p$. Thus, $P = \langle x, y \rangle$. Furthermore, as p is the smallest prime dividing $|P|$, any subgroup of order p is normal in P. So $\langle x \rangle, \langle y \rangle \trianglelefteq P$. So $P \cong \langle x \rangle \times \langle y \rangle$. As x, y have order p, $\langle x \rangle \times \langle y \rangle \cong \mathbb{Z}_p \times \mathbb{Z}_p$. The result follows. □

Remark: This proof is a more elegant way to demonstrate that a group of order p^2 for a prime p is Abelian. An alternate proof exists using the quotient $P/Z(P)$. We take representatives of P, project them down to $P/Z(P)$, operate in the quotient group, then lift back to P.

We next generalize Theorem 3.33.

Theorem 3.34. *Let p be prime, and let P be a p-group. Suppose $H \trianglelefteq P$, with $H \neq \{1\}$. Then $H \cap Z(P) \neq \{1\}$.*

Proof. Let P act on H by conjugation. Denote H^P as the set of fixed points under this action, and let $\mathcal{K}_1, \ldots, \mathcal{K}_r$ be the non-trivial conjugacy classes under this action. We have that:

$$|H| = |H^P| + \sum_{i=1}^{r} |\mathcal{K}_r|.$$

We note that as $H \leq P$, p divides $|H|$. Next, we note that $|\mathcal{K}_i| = [H : C_P(h_i)]$, where $h_i \in \mathcal{K}_i$ is arbitrary. As $\mathcal{K}_i$ is non-trivial, $|\mathcal{K}_i| > 1$. So by the Orbit-Stabilizer Theorem, p divides $|\mathcal{K}_i|$. Observe that $1 \in H^P$, so H^P is non-empty. Thus, p divides $|H^P|$. In particular, we note that:

$$\begin{aligned} H^P &= \{h \in H : ghg^{-1} = h, \text{ for all } g \in P\} \\ &= \{h \in H : gh = hg, \text{ for all } g \in P\} \\ &= H \cap Z(P). \end{aligned}$$

In particular, we have that as p divides $|H^P|$ and $|H^P| > 1$, that $H^P \neq 1$. The result follows. □

We now consider the case when the symmetry group acts on itself by conjugation. We obtain several important results. The first result we present shows that the permutation cycle type is preserved under conjugation. This observation was the key to breaking the Enigma cipher during World War II. The preservation of cycle type under conjugation yields a nice bijection between integer partitions of n and the conjugacy classes of S_n. Note that an integer partition is a sequence of non-decreasing positive integers that add up to n.

Proposition 3.26. *Let $\sigma, \tau \in Sym(n)$. The cycle decomposition of $\tau\sigma\tau^{-1}$ is obtained from σ by replacing each entry i in the cycle decomposition of σ with $\tau(i)$.*

Proof. Suppose $\sigma(i) = j$. As τ is a permutation, we consider the input $\tau(i)$ without loss of generality. So $\tau\sigma\tau^{-1}(\tau(i)) = \tau\sigma(i) = \tau(j)$. So while i, j appear consecutively in σ, $\tau(i)$ precedes $\tau(j)$ in $\tau\sigma\tau^{-1}$. □

We now formally define the cycle type of a permutation.

Definition 115 (Cycle Type)**.** Let $\sigma \in \text{Sym}(n)$ be the product of disjoint cycles of lengths $n_1, \ldots, n_k$ with $n_1 \leq n_2 \leq \ldots \leq n_k$ (including the 1-cycles), then the sequence of integers $(n_1, \ldots, n_k)$ is the *cycle type* of σ. Note that $n_1 + n_2 + \ldots + n_k = n$.

Remark: It is easy to see the bijection between integer partitions and cycle types. So it remains to be shown that all permutations of a given cycle type belong to the same conjugacy class.

Proposition 3.27. *Two elements of Sym(n) are conjugate in Sym(n) if and only if they have the same cycle type. The number of conjugacy classes of Sym(n) equals the number of partitions of n.*

Proof. If two permutations are conjugate in $\text{Sym}(n)$, then they have the same cycle type by Proposition 3.26. Conversely, suppose two permutations σ and τ have the same cycle type in $\text{Sym}(n)$. We construct a permutation γ such that $\tau = \gamma\sigma\gamma^{-1}$. We begin by ordering the cycles in the decompositions into disjoint cycles of σ and τ in non-decreasing order by length, including the 1-cycles. That is, we write:

$$\begin{aligned} \sigma &= \alpha_1 \cdots \alpha_k, \text{ and} \\ \tau &= \beta_1 \cdots \beta_k, \end{aligned}$$

where α_i and β_i have the same length. We write:

$$\begin{aligned} \alpha_i &= (\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{i\ell}), \text{ and} \\ \beta_i &= (\beta_{i1}, \beta_{i2}, \ldots, \beta_{i\ell}). \end{aligned}$$

Define the permutation γ by $\gamma(\alpha_{ij}) = \beta_{ij}$. From the proof of Proposition 3.26, $\gamma\alpha_i\gamma^{-1} = \beta_i$. So:

$$\begin{aligned} \gamma\sigma\gamma^{-1} &= \gamma(\alpha_1 \cdots \alpha_k)\gamma^{-1} \\ &= \prod_{i=1}^{k} (\gamma\alpha_i\gamma^{-1}) \\ &= \prod_{i=1}^{k} \beta_i \\ &= \tau. \end{aligned}$$

So σ and τ are conjugate, as desired. □

We illustrate the bijection in the case of Sym(5).

Example 128.

Partition of 5	**Representative of Conjugacy Class**
$1,1,1,1,1$	(1)
$1,1,1,2$	$(1,2)$
$1,1,3$	$(1,2,3)$
$1,4$	$(1,2,3,4)$
5	$(1,2,3,4,5)$
$1,2,2$	$(1,2)(3,4)$
$2,3$	$(1,2)(3,4,5)$

Example 129. Using the previous proposition and the Orbit-Stabilizer Lemma, we are able to compute the number of conjugates and centralizers for various permutations. We consider the case of an m cycle $\sigma \in \text{Sym}(n)$. Recall that all m cycles belong to the same conjugacy class as σ. We first compute the number of m-cycles in $\text{Sym}(n)$. We select m elements from $[n]$ which can be done in $\binom{n}{m}$ ways. Each set of m selements is then permuted in $m!$ ways. As cyclic rotations of a cycle yield the same permutation, we divide out by m to obtain $\binom{n}{m} \cdot (m-1)!$ cycles of length m in $\text{Sym}(n)$. This is the order of the orbit or conjugacy class containing σ.

By the Orbit-Stabilizer Lemma, we have $\binom{n}{m} \cdot (m-1)! = \dfrac{\text{Sym}(n)}{|C_G(\sigma)|}$, where $|\text{Sym}(n)| = n!$. We now compute $|C_G(\sigma)|$. Any permutation σ commutes with $\langle\sigma\rangle$. Additionally, σ commutes with of the permutations from which it is disjoint. There are $(n-m)!$ such permutations. By the Orbit-Stabilizer Lemma, $|C_G(\sigma)| = m \cdot (n-m)!$. Similar combinatorial analysis can be used to deduce the order of both the centralizers and conjugacy classes for other cycle types.

The integer partitions of $n \in \mathbb{N}$ can be enumerated using techniques from algebraic combinatorics, such as generating functions. Nicholas Loehr's *Bijective Combinatorics* text and Herbert Wilf's *Generatingfunctionology* text are good resources for further study on enumerating integer partitions.

3.4.2 Automorphisms of Groups

In this section, we study basic properties of automorphisms of groups. We have already seen examples of automorphisms, with the study of graphs. Analogously, for a group G, $\text{Aut}(G)$ denotes the automorphism group of G. The study of automorphisms provides additional information about the structure of a group and its subgroups. We begin by showing the action of conjugation induces automorphisms.

Theorem 3.35. *Let G be a group, and let $H \trianglelefteq G$. Then G acts by conjugation on H as automorphisms of H. In particular, the permutation representation of this action is a homomorphism from G into Aut(H) with kernel $C_G(H)$, with $G/C_G(H) \leq$ Aut(H).*

Proof. Let $g \in G$, and let $\sigma_g : h \mapsto ghg^{-1}$ be the permutation representation of g. As H is normal, each such σ_g is a bijection. It suffices to show that σ_g is a homomorphism. Let $h, k \in H$ and consider $\sigma_g(hk) = g(hk)g^{-1} = (ghg^{-1})(gkg^{-1}) = \sigma_g(h)\sigma_g(k)$. So $\sigma_g \in \text{Aut}(H)$. The kernel of this action are precisely the elements in g which induce the trivial action, which is equivalent to the kernel being $C_G(H)$. We apply the First Isomorphism Theorem to deduce that $G/C_G(H) \leq \text{Aut}(H)$. □

Theorem 3.35 has a couple nice corollaries to tedious homework problems from the introductory group theory material. In particular, it follows immediately that conjugate elements and conjugate subgroups have the same order.

Corollary 3.27.1. *Let K be a subgroup of the group G. Then $K \cong gKg^{-1}$ for any $g \in G$. Conjugate elements and conjugate subgroups have the same order.*

Proof. We apply Theorem 3.35, using $H = G$ as G is normal in itself. The result follows immediately. □

Corollary 3.27.2. *For any subgroup H of a group G, the quotient group $N_G(H)/C_G(H) \leq$ Aut(H). In particular, $G/Z(G) \leq$ Aut(G).*

Proof. We apply Theorem 3.35 to $N_G(H)$ acting on H by conjugation to deduce that $N_G(H)/C_G(H) \leq \text{Aut}(H)$. As $G = N_G(G)$ and $C_G(G) = Z(G)$, we have that $G/Z(G) \leq \text{Aut}(G)$ by the previous case. □

Definition 116 (Inner Automorphisms)**.** Let G be a group, and let $g \in G$. The *inner automorphisms* of G are $\text{Inn}(G) \cong G/Z(G)$. The *outer automorphisms* of G are $Out(G) \cong \text{Aut}(G)/\text{Inn}(G)$.

Remark: Note that $\text{Inn}(G)$ is normal in $\text{Aut}(G)$, so any inner automorphism σ of the group G satisfies $g \circ \sigma(a)g^{-1} = \sigma(g(a))$ for any $g \in \text{Aut}(G)$. The group $\text{Out}(G)$ measures how far away $\text{Aut}(G)$ is from consisting only of inner automorphisms.

We conclude with a final fact about cyclic groups.

Proposition 3.28. *Let $G \cong \mathbb{Z}_n$. Then Aut$(G) \cong \mathbb{Z}_n^{\times}$.*

Proof. There are $\phi(n)$ generators of $\mathbb{Z}_n$, where $\phi(n)$ denotes Euler's Totient Function. So for every a such that $\gcd(a, n) = 1$, the map $\sigma_a : x \mapsto x^a$ is an automorphism. The map sending $\sigma_a \mapsto \overline{a}$ is a surjective map from $\text{Aut}(\mathbb{Z}_n) \to \mathbb{Z}_n^{\times}$. Now observe that $\sigma_a \circ \sigma_b(x) = (x^b)^a = x^{ab} = \sigma_{ab}(x)$, so the map sending $\sigma_a \mapsto \overline{a}$ is a homomorphism. The kernel of this homomorphism is $\{1\}$; so by the First Isomorphism Theorem, $\text{Aut}(\mathbb{Z}_n) \cong \mathbb{Z}_n^{\times}$. □

3.4.3 Sylow's Theorems

The Sylow Theorems are a stronger partial converse to Lagrange's Theorem than Cauchy's Theorem. More importantly, they provide an important set of combinatorial tools to study the structure of finite groups and are the high point of a senior algebra course. Standard proofs of the Sylow's First Theorem usually proceed by induction, leveraging the Well-Ordering Principle in the background. We instead offer a combinatorial proof using group actions, which is far more enlightening and elegant. To do this, we need a result from combinatorial number theory known as Lucas' Congruence for Binomial Coefficients. We begin with a couple helpful lemmas, which we will need to prove Lucas' Congruence for Binomial Coefficients.

Lemma 3.8. *Let $j, m \in \mathbb{N}$ and p be prime. Then:*

$$\binom{m+p}{j} \equiv \binom{m}{j} + \binom{m}{j-p} \pmod{p}.$$

Proof. Let $\mathbb{Z}_p$ act on $Y = \binom{[m+p]}{j}$ sending $S \mapsto gS = \{g \cdot s : s \in S\}$. The orbits under this action partition Y. Every orbit has order 1 or order p, as p is prime. So $|Y|$ is congruent modulo p to the number of orbits M of order 1. We show $M = \binom{m}{j} + \binom{m}{j-p}$. The orbits of order 1 are in the kernel of the action; that is, the sets $S \in Y$ such that $gS = S$ for all $g \in \mathbb{Z}_p$. It suffices to count the number of sets $S \in Y$ such that the generator $h = (1, 2, \dots, p) \in \mathbb{Z}_p$ fixes S. Observe that $h(x) = x$ for all $x > p$. We note that there are $\binom{m}{j}$ sets such that $S \cap [p] = \emptyset$. Each such set S is fixed under the action of $\mathbb{Z}_p$. If instead $S \cap [p] \neq \emptyset$, it is necessary that $[p] \subset S$. This leaves $j - p$ remaining choices for elements in S, which must be chosen from $\{p+1, \dots, m+p\}$. So there are $\binom{m}{j-p}$ such selections. By rule of sum, these cases are disjoint, so $M = \binom{m}{j} + \binom{m}{j-p}$. This completes the proof. □

Lemma 3.9. *Let p be prime. Let $a, c \in \mathbb{N}$ and $0 \leq b, d < p$. Then $\binom{ac+b}{cp+d} \equiv \binom{a}{c}\binom{b}{d} \pmod{p}$.*

Proof. The proof is by induction on a. When $a = 0$ and $c > 0$, both sides of the congruence are 0. If $a = c = 0$, then both sides of the congruence are $\binom{b}{d}$. Now suppose this result holds up to a given a, and for all b, c, d. We prove true for the $a + 1$ case. Consider $\binom{(a+1)p+b}{cp+d} = \binom{(ap+b)+p}{p+d}$. We apply Lemma 3.8 with $m = ap + b$ and p to obtain the following:

$$\begin{aligned}\binom{(ap+b)+p}{p+d} &\equiv \binom{ap+b}{cp+d} + \binom{ap+b}{(c-1)p+d} \pmod{p} \\ &\equiv \binom{a}{c}\binom{b}{d} + \binom{a}{c-1}\binom{b}{d} \pmod{p},\end{aligned}$$

where the last equality from the inductive hypothesis. We now apply the binomial identity that $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$ and factor the $\binom{b}{d}$ to obtain:

$$\binom{a}{c}\binom{b}{d} + \binom{a}{c-1}\binom{b}{d} \equiv \left(\binom{a}{c} + \binom{a}{c-1}\right)\binom{b}{d} \equiv \binom{a+1}{c}\binom{b}{d} \pmod{p}.$$

The result follows. □

With Lemmas 3.8 and 3.9 in tow, we prove Lucas' Congruence for Binomial Coefficients.

Theorem 3.36 (Lucas' Congruence for Binomial Coefficients)**.** *Let p be prime, and let $k, n \in \mathbb{N}$ with $k \leq n$. Consider the base-p expansions $n = \sum_{i \geq 0} n_i p^i$ and $k = \sum_{i \geq 0} k_i p^i$, where $0 \leq n_i, k_i < p$. Then:*

$$\binom{n}{k} \equiv \prod_{i \geq 0} \binom{n_i}{k_i} \pmod{p},$$

where $\binom{0}{0} = 1$ and $\binom{a}{b} = 0$ whenever $b > a$.

Proof. The proof is by induction on n. When $k > n$, then $k_i > n_i$ for some i. So both sides of the congruence are 0. We now consider the case when $k \leq n$. The result holds when $n \in \{0, \dots, p-1\}$ as $n_0 = n, k_0 = k$ and for all $i > 0$ we have $n_i = k_i = 0$. Now suppose the result holds true up to some arbitrary $n - 1 \geq p - 1$. We prove true for the n case. By the division algorithm, we write $n = ap + n_0$ and $k = bp + k_0$ where $n_0, k_0 \in \{0, \dots, p-1\}$. We write $a = \sum_{i \geq 0} n_{i+1} p^i$ and $c = \sum_{i \geq 0} k_{i+1} p^i$ in base p. We apply Lemma 3.9 to obtain that:

$$\binom{n}{k} \equiv \binom{ap}{bp}\binom{n_0}{k_0} \pmod{p}.$$

Applying the inductive hypothesis to $\binom{ap}{bp}$, we obtain that:

$$\binom{ap}{bp}\binom{n_0}{k_0} \equiv \binom{n_0}{k_0} \prod_{i \geq 1} \binom{n_i}{k_i} \equiv \prod_{i \geq 0} \binom{n_i}{k_i} \pmod{p}.$$

The completes the proof. □

Lucas' Congruence for Binomial Coefficients provides a nice corollary, which we will need to prove Sylow's First Theorem.

Corollary 3.28.1. *Let $a, b \in \mathbb{Z}^+$, and let p be a prime that does not divide b. Then p does not divide $\binom{p^a b}{p^a}$.*

Proof. We write $b = \sum_{i \geq 0} b_i p^i$ in base p. The base p expansion of $p^a b = \ldots b_3 b_2 b_1 b_0 000 \ldots 0$, and the base p expansion of $p^a = 1 \ldots 00000$. Without loss of generality, suppose $b_0 \neq 0$. By Lucas' Congruence for Binomial Coefficients, we have:

$$\binom{p^a b}{p^a} \equiv \binom{b_0}{1} \equiv b_0 \neq 0 \pmod{p}.$$

□

We now introduce a couple definitions before examining the Sylow Theorems.

Definition 117. Let G be a finite group, and let p be a prime divisor of $|G|$.

(A) A group of order p^α, for $\alpha > 0$, is called a *p-group*. Subgroups of p-groups are called p-subgroups.

(B) If $|G| = p^\alpha m$, where p does not divide m, then a subgroup of order p^α is called a Sylow p-subgroup of G. The set of Sylow p-subgroups of G is denoted $\text{Syl}_p(G)$, and $n_p(G) := |\text{Syl}_p(G)|$. When there is no ambiguity, we may simply right n_p instead of $n_p(G)$.

The Sylow Theorems provide combinatorial tools to count the number of Sylow p-subgroups, as well as characterize structural results. In particular, the Sylow Theorems give us the result that $n_p = 1$ if and only if unique Sylow p-subgroup is normal in G. This result helps us determine if a finite group is simple; that is, a group whose only normal subgroups are 1 and G. We begin with Sylow's First Theorem, which provides for the existence of Sylow p-subgroups for every prime divisor p of a finite group $|G|$. So Sylow's First Theorem is a stronger partial coverse to Lagrange's Theorem than Cauchy's Theorem.

Theorem 3.37 (Sylow's First Theorem)**.** *Let G be a finite group, and let p be a prime divisor of $|G|$. We write $|G| = p^\alpha m$, where $\alpha \in \mathbb{N}$ and p does not divide m. Then there exists a $P \leq G$ of order p^α. That is, $Syl_p(G) \neq \emptyset$.*

Proof. Let $X = \binom{G}{p^\alpha}$, so $|X| = \binom{p^\alpha m}{p^\alpha}$. Now let G act on X by left multiplication. So for $S \in X$, $gS = \{gs : s \in S\}$. By Lucas' Congruence for Binomial Coefficients, we note p does not divide $|X|$. So there exists some $T \in X$ such that $|\mathcal{O}(T)|$ is not divisible by p. By the Orbit-Stabilizer Lemma, $|\mathcal{O}(T)| = |G|/|\text{Stab}(T)| = p^\alpha m/|\text{Stab}(T)|$. As p does not divide $|\mathcal{O}(T)|$, $|\text{Stab}(T)| = cp^\alpha$ for some c that divides m. It suffices to show $c = 1$. Let $t \in T$ and consider $\text{Stab}(T)t = \{ht : h \in \text{Stab}(T)\}$, which is a subset of T. So $|\text{Stab}(T)| = |\text{Stab}(T)t| \leq |T| = p^\alpha$. Thus, $\text{Stab}(T) \in \text{Syl}_p(G)$. □

Prior to introducing Sylow's Second Theorem, we prove a lemma known as the p-group Fixed Point Theorem. We leverage this the p-group Fixed Point Theorem in proving both Sylow's Second and Third Theorems.

Theorem 3.38 (p-group Fixed Point Theorem)**.** *Let p be a prime, and let G be a finite group of order p^α for $\alpha > 0$. Let G act on a set X, and let S be the set of fixed points under this action. Then $|G| \equiv |S| \pmod{p}$. In particular, if p does not divide $|X|$, then $|S| \neq \emptyset$.*

Proof. Let $x \in X \setminus S$. Then $\text{Stab}(x)$ is a proper subgroup of G. Thus, p divides $[G : \text{Stab}(x)]$. Recall that the orbits partition X. Let $x_1, \ldots, x_k \in X \setminus S$ be representatives of the non-fixed point orbits. As the orbits partition X and by the Orbit-Stabilizer Lemma, we have:

$$|X| = |S| + \sum_{i=1}^{k} [G : \text{Stab}(x_i)].$$

As $\sum_{i=1}^{k} [G : \text{Stab}(x_i)]$ is divisible by p, we have $|X| \equiv |S| \pmod{p}$. If p does not divide $|X|$, then $|S| \neq 0 \pmod{p}$. So $S \neq \emptyset$ in this case. □

Sylow's Second Theorem follows as a consequence of the p-group Fixed Point Theorem. We show first that every p-subgroup is contained in a Sylow p-subgroup of G. This is done using the action of conjugation. Intuitively, a p-subgroup of G cannot be contained in a subgroup $H \leq G$ where $|H|$ divides m. This is a consequence of Lagrange's Theorem. In particular, Lagrange's Theorem implies that every subgroup of a Sylow p-subgroup is a p-subgroup of G. Sylow's Second Theorem provides a full converse to this statement. We then show that every pair of Sylow p-subgroups are conjugate, which follows from the fact that every p-subgroup of G is contained in a Sylow p-subgroup of G.

Theorem 3.39 (Sylow's Second Theorem)**.** *Let G be a finite group, and let p be a prime divisor of $|G|$. We write $|G| = p^\alpha m$, where $\alpha \in \mathbb{N}$ and p does not divide m. If P is a Sylow p-subgroup of G and Q is a p-subgroup of G, then there exists a $g \in G$ such that $Q \leq gPg^{-1}$. In particular, any two Sylow p-subgroups of G are conjugate .*

Proof. Let $P \in \text{Syl}_p(G)$. We consider the left cosets of G/P. Let Q act on G/P by left-multiplication. Observe that p does not divide $[G : P] = |G/P|$. By the previous theorem, there exists a fixed point of this action. Let gP be such a fixed point. So for every $q \in Q$, $qgP = gP$, so $g^{-1}qgP = P$. That is, $g^{-1}Qg \subset P$; or equivocally, $Q \subset gPg^{-1}$. Thus, $Q \leq gPg^{-1}$ for some g, as desired. To show that all Sylow p-subgroups are conjugate, we utilize the above argument setting Q to be a Sylow p-subgroup of G. □

Sylows's Third Theorem provides us a way to determine the number of Sylow p-subgroups in a finite group G. This is particularly useful in deciding if a finite group has a normal subgroup of given prime power order. In turn, we have a combinatorial tool to help us decide if a finite group is simple. Before proving Sylow's Third Theorem, we introduce a helpful lemma.

Lemma 3.10. *Let G be a finite group, and let H be a p-subgroup of G. Then $[N_G(H) : H] \equiv [G : H] \pmod{p}$.*

Proof. Let H act on the set of left cosets G/H by left multiplication. The set of fixed points are of the form gH where $hgH = gH$ for all $h \in H$. This is equivalent to $g^{-1}hg \in H$ for all $h \in H$. This is equivalent to $g^{-1}Hg = H$. So if gH is a fixed point under this action, then $g \in N_G(H)$. This is equivalent to $gH \in N_G(H)/H$. By the p-group fixed point theorem, $[G : H] = |G/H| \equiv |N_G(H)/H| \pmod{p}$. It follows immediately that $[N_G(H) : H] \equiv [G : H] \pmod{p}$. □

Theorem 3.40 (Sylow's Third Theorem)**.** *Let G be a finite group, and let p be a prime divisor of $|G|$. We write $|G| = p^\alpha m$, where $\alpha \in \mathbb{N}$ and p does not divide m. The number of Sylow p-subgroups of G, $n_p \equiv 1 \pmod{p}$. Furthermore, $n_p = [G : N_G(P)]$ for any $P \in Syl_p(G)$. So n_p divides m.*

Proof. By Sylow's Second Theorem, we have that all Sylow p-subgroups are conjugate. So G acts transitively on $\text{Syl}_p(G)$ by conjugation. Now $\text{Stab}(P) = N_G(P)$, for any $P \in \text{Syl}_p(G)$. So by the Orbit-Stabilizer Theorem, $n_p = [G : N_G(P)]$ for any $P \in \text{Syl}_p(G)$. In particular, we have:

$$m = [G : P] = [G : N_G(P)] \cdot [N_G(P) : P] = n_p \cdot [N_G(P) : P].$$

So n_p divides m. Now by Lemma 3.10, $m = [G : P] \equiv [N_G(P) : P] \pmod{p}$. As $m = n_p \cdot [N_G(P) : P]$, we have that $m \cdot n_p \equiv m \pmod{p}$. As p is prime and $n_p > 0$, $n_p \equiv 1 \pmod{P}$. □

The Sylow Theorems have a nice corollary, which allows us to characterize normal Sylow p-subgroups.

Corollary 3.28.2. *A Sylow p-subgroup P in the finite group G is normal in G if and only if $n_p = 1$.*

Proof. Suppose first $P \trianglelefteq G$. Then $N_G(P) = G$. As all Sylow p-subgroups are conjugate, the conjugacy class of P contains only P. This is equivalent to $n_p = 1$. □

3.4.4 Applications of Sylow's Theorems

The Sylow Theorems can be leveraged to provide deep insights into the structure of finite groups, particularly as it pertains to the existence of normal subgroups. This in turn allows us to better understand the structures of individual groups, large classes of finite groups, and to classify groups of a given order. We consider some examples. Let G be a finite group of order $p^\alpha m$ where p is a prime that does not divide m. Informally, if p^α is sufficiently large, then we have a unique Sylow p-subgroup

Proposition 3.29. *Let p be prime, $r \in \mathbb{Z}^+$, and $m \in [p-1]$. Let G be a group of order mp^r. Then G is not simple (that is, G has a proper normal subgroup).*

Proof. By Sylow's Third Theorem, $n_p \equiv 1 \pmod p$ and n_p divides m. Since $m < p$, this forces $n_p = 1$. So $P \in \text{Syl}_p(G)$ is a proper normal subgroup of G. $\square$

We now examine groups of order pq, where p and q are primes and $p < q$.

Proposition 3.30. *Let G be a group of order pq, where p and q are primes with $p < q$. Let $P \in Syl_p(G)$ and $Q \in Syl_q(G)$. Then $Q \trianglelefteq G$. If $q \not\equiv 1 \pmod p$, then $P \trianglelefteq G$ as well and G is cyclic.*

Proof. By Sylow's Third Theorem, $n_q \equiv 1 \pmod q$ and n_q divides p. So $n_q \in \{1, p\}$. As $p < q$, $n_q = 1$, so $Q \trianglelefteq G$. Now suppose $q \not\equiv 1 \pmod p$. As $n_p | q$, have that $n_p \in \{1, q\}$ as q is prime. As $q \neq 1 \pmod p$ by assumption and $n_p \equiv 1 \pmod p$, we have that $n_p = 1$. So $P \trianglelefteq G$. As $|P| = p$, P is cyclic. By Theorem 3.35, $G/C_G(P) \leq \text{Aut}(P)$. As p is prime and P is cyclic, $|\text{Aut}(P)| = p-1$. By Lagrange's Theorem, neither p nor q divide $p-1$. So $C_G(P) = G$, which implies that $P \leq Z(G)$. Now G contains elements $g \in P$ of order p and $h \in Q$ of order q. As $P \leq Z(G)$, $gh = hg$. So $|gh| = pq$. We conclude that $G \cong \mathbb{Z}_{pq}$. $\square$

We next show that every group of order 30 has a subgroup isomorphic to $\mathbb{Z}_{15}$. Observe that if G is a group of order 30, then $[G : \mathbb{Z}_{15}] = 2$, so $\mathbb{Z}_{15}$ is necessarily a normal subgroup of G.

Proposition 3.31. *Let G be a group of order 30. Then $\mathbb{Z}_{15} \trianglelefteq G$.*

Proof. Note that $|G| = 2 \cdot 3 \cdot 5$. By Sylow's Third Theorem, we have that $n_3 \equiv 1 \pmod 3$ and $n_3 | 10$. So $n_3 \in \{1, 10\}$. By similar argument, $n_5 \in \{1, 6\}$. Suppose to the contrary that no Sylow-3 or Sylow-5 subgroup is normal in G. Then $n_3 = 10$ and $n_5 = 6$. Each Sylow-5 subgroup contains four non-identity elements, and each Sylow-3 subgroup contains two non-identity elements. This provides 20 elements of order 3 and 24 elements of order 4. However $20 + 24 > 30$, a contradiction. So there exists a normal Sylow-3 subgroup or normal Sylow-5 subgroup in G. Let $P \in \text{Syl}_3(G)$ and $Q \in \text{Syl}_5(G)$. By Corollary 3.21.1, $PQ \leq G$. As $[G : PQ] = 2$, $PQ \trianglelefteq G$. So by Proposition 3.30, $PQ \cong \mathbb{Z}_{15}$. $\square$

We next show that there are no simple groups of order 12. We will use this result to study simple groups of order 60. In particular, this result will help us show that A_5 is simple.

Proposition 3.32. *There is no simple group of order 12.*

Proof. By Sylow's Third Theorem, $n_3 \equiv 1 \pmod 3$ and $n_3 \in \{1, 4\}$. Similarly, $n_2 \equiv 1 \pmod 2$ and $n_2 \in \{1, 3\}$. If $n_3 = 1$, then we are done. Suppose instead that $n_3 = 4$. As every Sylow-3 subgroup is normal, each Sylow-3 subgroup has trivial intersection. This provides for 8 elements of order 3 and the identity. So necessarily, there exists one Sylow-2 subgroup, which has order 4. This accounts for the remaining three elements of order 2 or order 4. So there exists a non-trivial normal subgroup in a group of order 12. $\square$

Proposition 3.33. *If G is a group of order 60 and G has more than one Sylow-5 subgroup, then G is simple.*

Proof. Suppose to the contrary that G has more than one Sylow-5 subgroup and G is not simple. Let $H \trianglelefteq G$ be a proper subgroup of G, with $H \neq 1$. By Sylow's Third Theorem, $n_5 = 6$. Let $P \in \text{Syl}_P(G)$. So $|N_G(P)| = 10$ as $n_5 = [G : P] = 6$. Now suppose $5 \big| |H|$. Then H contains a Sylow-5 subgroup Q of G. As H is normal, H necessarily contains all the conjugates of Q, which are the 6 Sylow-5 subgroups of G. So $|H| > 25$, which implies $|H| = 30$. However, by Proposition 3.31, H contains a unique Sylow-5 subgroup, which is in turn a Sylow-5 subgroup of G. This contradicts the assumption that $n_5 = 6$. So 5 does not divide H.

If $|H| = 6$, then there is a single Sylow-3 subgroup Q in H which is also a Sylow-3 subgroup of G. As H contains all the conjugates of Q, $Q \trianglelefteq G$. So in this case, G is not simple.

Now by Proposition 3.32, if $|H| = 12$, then H contains a normal subgroup which is also a normal Sylow subgroup of G. So without loss of generality, we have a normal subgroup of K of G with order $|K| \in \{3, 4\}$. So $|G/K| \in \{15, 20\}$. By the first paragraph, there exists $\overline{P} \trianglelefteq |G/K|$ with $|\overline{P}| = 5$. Let $\pi : G \to G/K$ be the natural projection homomorphism sending $g \mapsto gK$. By the Lattice Isomorphism Theorem, $P := \pi^{-1}(\overline{P}) \trianglelefteq G$. As $\overline{P}$ has order 5, $|P|$ is necessarily divisible by 5. However, we showed in the first paragraph that G does not have a normal subgroup whose order is divisible by 5. So G is necessarily simple. $\square$

Corollary 3.33.1. *A_5 is simple.*

Proof. Observe that $\langle(1,2,3,4,5)\rangle$ and $\langle(1,3,2,4,5)\rangle$ are two distinct Sylow-5 subgroups of A_5. So by Proposition 3.34, A_5 is simple. $\square$

We conclude with the following remark. In many of these counting arguments, we used the fact that two Sylow-p subgroups intersected trivially. This holds true for cyclic subgroups; however, when p^α for $\alpha \geq 2$, two Sylow-p subgroups may have non-trivial intersection. In these cases, the problems are not as amenable to counting arguments.

3.4.5 Algebraic Combinatorics- Pólya Enumeration Theory

TODO

4 Turing Machines and Computability Theory

In this section, we explore the power and limits of computation without regards to resource usage. That is, we seek to study which problems computers can and cannot solve, given unlimited time and space. More succinctly, the goal is to provide a model of computation powerful enough to be representative of an algorithm. The primitive automata in previous sections motivate this problem. Finite state automata compute memoryless algorithms. While regular languages are quite interesting and useful, finite state automata fail to decide context free languages such as $L_1 = \{0^n 1^n : n \in \mathbb{N}\}$. To this end, we add an infinite stack to the finite state automaton, where only the head of the stack may be accessed. We refer to this modified finite state automaton as a *pushdown automaton*, which accepts precisely context-free languages. We again find a language beyond the limits of pushdown automata- $L_2 = \{0^n 1^n 2^n : n \in \mathbb{N}\}$, which does not satisfy the Pumping Lemma for Context-Free Languages.

However, it is quite simple to design an algorithm to verify if an input string is of the form prescribed by L_1 or L_2. In fact, this would be a reasonable question in an introductory programming class. So clearly, it is quite feasible to decide if a string is in L_1 or L_2. Thus, both of our models of computation, the finite state automata and pushdown automata, are unfit to represent an algorithm in the most general sense. To this end, we introduce a Turing Machine, the model of computation which serves as the litmus test for which problems are solvable by computational means.

It is also important to note that there are numerous models of computation that are vastly different from the Turing Machine; but with the exception of hypercomputation model, none are more powerful than the Turing Machine. The Church-Turing Thesis conjectures that no model of computation is more powerful than the Turing Machine. As hypercomputation is not thus far physically realizable, the Church-Turing Thesis remains essentially an open problem.

4.1 Standard Deterministic Turing Machine

The standard deterministic Turing machine shares many similarities with the finite state automaton. Just like the finite state automaton, the Turing Machine solves decision problems; that is, it attempts to decide if a string is in a given language. Furthermore, both have a finite set of states. The next state of each machine is determined by the given character being parsed and the current state. Unlike a finite state automaton though, a Turing Machine can both read and write to the tape head. Furthermore, the Turing Machine also has unlimited memory to the right with a fixed end at the left, and the tape head can move both left and right. This allows us to parse characters multiple times and develop the notion of iteration in our computations. Lastly, the Turing Machine has explicit accept and reject states, which take effect immediately upon being reached. Formally, we define the standard Turing Machine as follows.

Definition 118 (Deterministic Turing Machine). A 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 $\Sigma \subset \Gamma$.

- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L, R}\}$ 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 $q_{\text{reject}} \neq q_{\text{accept}}$.

Let's now unpack the Turing Machine some more. Conceptually, a standard Turing Machine starts with an input string, which is written to an initially blank tape starting at the far left cell. It then executes starting at the initial state q_0, transitioning to other states as defined by the function δ, based on the current state and input from the given tape cell. If evaluating this string in such a manner results in the Turing Machine reaching its accepting halting state q_{accept}, then the Turing Machine is said to accept the input string. If the Turing Machine does not visit q_{accept}, then it does not accept the given input string. However, if it does not explicitly visit q_{reject}, then the Turing Machine does not reject the input string; rather, it enters into an infinite loop. The language of a Turing Machine M, $L(M)$, is the set of strings the Turing Machine M accepts. We introduce two notions of language acceptance.

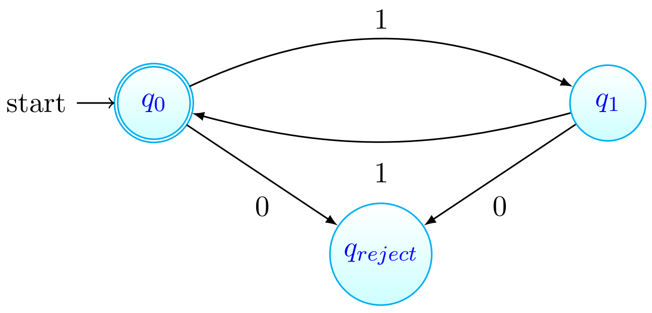
Definition 119 (Recursively Enumerable Language)**.** A language L is said to be *recursively enumerable* if there exists a deterministic Turing Machine M such that $L(M) = L$. Note that if $\omega \not\in L$, the machine M need not halt on ω.

Definition 120 (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: 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.

We now consider an example of a Turing Machine.

Example 130. Let $\Sigma = \{0, 1\}$ and let $L = \{1^{2k} : k \in \mathbb{N}\} = (11)^*$, so L is regular. A finite state automaton 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 context-free language.

Example 131. Let $\Sigma = \{0, 1\}$ and let $L = \{0^n 1^n : n \in \mathbb{N}\}$. So L is context-free. We omit the construction of a pushdown automaton, but simply 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: 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.*"

We now introduce the notion of a configuration and provides a concise representation of the Turing Machine's state, tape head position, and the string written to the tape. Aside from providing a concise representation of the Turing Machine, configurations are important in studying how Turing Machines work. In particular, certain results in space complexity are derived by enumerating the possible configurations of a Turing Machine on an arbitary input string. We formally define a Turing configuration below.

Definition 121 (Turing Machine Configuration)**.** Let M be a Turing Machine run on the string s. A *Turing Machine Configuration* is a string $\omega \in \Gamma^* Q \Gamma^*$, where Q is the current state of M, which we overlay on the character of s highlighted by the tape head. The remaining characters in ω are the characters of the input string s which M is parsing. The *start configuration* of M is $q_0 s_1 \ldots s_n$ (where $n = |s|$). The *accept configuration* is a Turing Machine configuration where the state is q_{accept}, while in a *rejecting configuration* is a configuration where the state is q_{reject}. The accepting and rejecting configurations are both halting configurations.

Example 132. Recall the Turing Machine in Example 130. We consider the input string 1111. The initial configuration is $q_0 111$. The subsequent configuration is $1q_1 11$.

This example motivates the *yields relation*, which enables us to textually represent a sequence of Turing computations on a given input string.

Definition 122 (Yields Relation)**.** Let M be a Turing Machine parsing the input string ω. The *yields relation* is a binary relation on $\Gamma^*Q\Gamma^*$. We say that the configuration C_i *yields* the configuration C_{i+1} if C_{i+1} can be reached from a single step (or invocation of the transition function) from C_i. We denote this relation as $C_i \vdash C_{i+1}$.

Example 133. In running the Turing Machine from Example 130 on 1111, we have the sequence of configurations: $q_0111 \vdash 1q_111$. Similarly, $1q_111 \vdash 11q_01$. We then have $11q_01 \vdash 111q_1$, and in turn $111q_1 \vdash 1111q_{\text{accept}}$, where $1111q_{\text{accept}}$ is an accepting configuration.

4.2 Variations on the Standard Turing Machine

In automata theory, one seeks to understand the robustness of a model of computation with respect to variation. That is, does the introduction of nuances such as non-determinism or multiple tapes allow for a more powerful machine? Recall that deterministic and non-deterministic finite state automata are equally powerful, as they each accept precisely regular languages. When considering context-free languages, we see that non-deterministic pushdown automata are strictly more powerful than deterministic pushdown automata. 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 123 (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 4.1. *A language is recursively enumerable (decidable) if and only if some multitape Turing Machine accepts (decides) it.*

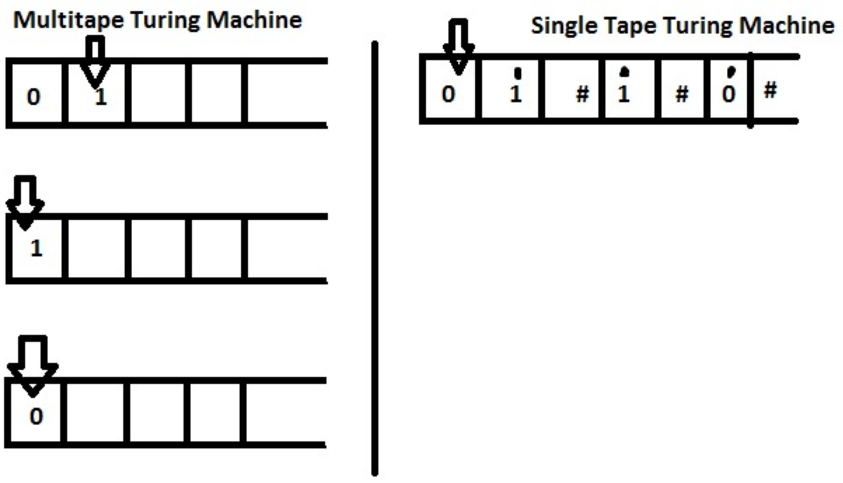
Proof. We begin by showing that the multitape Turing Machine model is at least as power as the standard deterministic Turing Machine model. Clearly, a standard deterministic Turing Machine is a 1-tape Turing Machine. So every language accepted (decided) by a standard deterministic Turing Machine is also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

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.

Definition 124 (Non-Deterministic Turing Machine)**.** A non-deterministic Turing Machine is defined identically as a standard deterministic Turing Machine, but with the transition function of the form:

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$$

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 4.2. *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 4.1, 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. □

4.3 Turing Machine Encodings

TODO

4.4 Chomsky Heirarchy and Some Decidable Problems

Thus far, we have introduced the Turing Machine as a model of computation and studied it from the perspective of automata theory. The goal of computability theory is to study the power and limits of computation. In this section, we explore problems that Turing Machines can effectively solve; that is, decidable languages.

Thus far, we have several classes of languages: regular, contex-free, decidable, and recursively enumerable languages. We have three relations that are easy to see:

1. Every regular language is context-free.
2. Every regular language is decidable.
3. Every decidable language is recursively enumerable.

These relationships are captured by the *Chomsky Heirarchy*, named for linguist Noam Chomsky. The Chomsky Heirarchy contains five classes of formal languages, with a strict increasing subset relation between them. Each class of formal language is characterized by the class of machines deciding them (or equivocally, the class of grammars generating them). The missing class of language is the set of context-sensitive languages. We mention them here for completeness, but will not explore them much further. Context-sensitive languages are accepted by linear bounded automata, which informally are Turing Machines with finite tape heads. It is thus easy to see that every context-sensitive language is recursively enumerable. In fact, every context-sensitive language is also decidable. We also note that every context-free language is also context-sensitive. So formally, we have the following relationships.

(a) Every regular language is context-free.

(b) Every context-free language is context-sensitive.

(c) Every context-sensitive language is decidable.

(d) Every decidable language is recursively enumerable.

In order to see that every regular language is decidable, we simply use a Turing Machine to simulate a finite state automaton. Given a regular language L, we construct a deterministic Turing Machine to decide L quite easily. Let D be the DFA accepting L. Without loss of generality, suppose D has precisely one accept state. We let M be the corresponding Turing Machine, with $Q_M = Q_D, \Sigma_D = \Sigma_D$; and for each $((q_i, a), q_j) \in \delta_D$, we include $((q_i, a), (q_j, a, L)) \in \delta_M$. So M simulates D, and $L(M) = L$. We rephrase this notion with the following theorem, which will be of use later.

Theorem 4.3. *Let $A_{DFA} = \{\langle D, w\rangle : D$ is a DFA that accepts $w\}$. A_{DFA} is decidable.*

Proof. We construct a Turing Machine M to decide A_{DFA} as follows. On input $\langle D, w\rangle$, M simulates D on w. M accepts $\langle D, w\rangle$ if and only if D accepts w. As every FSM is a decider, it follows that M is also a decider. So M decides A_{DFA}. □

Remark: We similarly define $A_{\text{NFA}} = \{\langle N, w\rangle : N \text{ is an NFA that accepts } w\}$. A_{NFA} is decidable. We apply the NFA to DFA algorithm, then utilize the decider for A_{DFA}.

We use a similar argument to decide if a regular expression recognizes a given string. Using a programmer's intuition, we utilize existing algorithms and theory developed in the exposition on regular languages.

Theorem 4.4. *Let $A_{REX} = \{\langle R, w\rangle : R$ is a regular expression that matches the string $w\}$. A_{REX} is decidable.*

Proof. We construct a Turing Machine M to decide A_{REX} as follows. M begins by converting R to an ϵ-NFA N using Thompson's Construction Algorithm. M then converts N to an NFA N' without ϵ transitions using the procedure outlined in our exposition on automata theory. From there, M simulates the decider S for A_{NFA} on $\langle N', w\rangle$ and accepts if and only if S accepts; and rejects if and only if S rejects. So M decides A_{REX}. □

We next consider two important problems: (1) Given a DFA D, is $L(D) = \emptyset$; and (2) Given two DFAs A and B, does $L(A) = L(B)$? Regular languages are decidable; so for every $w \in \Sigma^*$, some Turing Machine decides if w is in the corresponding regular language L. However, for other classes of decidable languages (such as context-sensitive languages), it is undecidable whether the corresponding automaton accepts no string, which makes testing for language equality an undecidable problem for that class of language. With this in mind, we proceed with our next results:

Theorem 4.5. *Let $E_{DFA} = \{\langle D\rangle : D$ is a DFA and $L(D) = \emptyset\}$. E_{DFA} is decidable.*

Proof. We construct a Turing Machine M to decide E_{DFA} as follows. M begins by labeling the start state. Then while no new states have been marked, we mark any state with an incoming transition from an already marked state. D accepts some string if and only if some accept state was marked. So M accepts if no accept state has been reached, and rejects otherwise. □

We show next that testing for language equality is decidable, provided we have two regular languages. Recall that $L_1 = L_2$ if and only if $L_1 \triangle L_2 = \emptyset$. In order to prove this, we leverage closure properties of regular langauges (and the FSMs constructed in the proofs of these properties) to construct a DFA recognizing $L_1 \triangle L_2$, then defer to the decider for E_{DFA}.

Theorem 4.6. *Let $EQ_{DFA} = \{\langle A, B\rangle : A, B$ are DFAs and $L(A) = L(B)\}$. EQ_{DFA} is decidable.*

Proof. We construct a Turing Machine M to decide EQ_{DFA}. Recall that:

$$L(A) \triangle L(B) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

As regular languages are closed under union, intersection, and complementation, we construct a DFA for C using the constructions given in the proofs of these closure properties. So $L(C) = \emptyset$ if and only if $L(A) \triangle L(B) = \emptyset$. And $L(A) = L(B)$ if and only if E_{DFA} accepts $\langle C\rangle$. So M accepts $\langle A, B\rangle$ if and only if E_{DFA} accepts $\langle C\rangle$, and M rejects otherwise. So M decides EQ_{DFA}. □

We now provide analogous results for context-free languages as for regular languages. These results culminate with the following result: every context-free language is decidable. We will need a couple facts about context-free languages:

- Every context-free language has a context-free grammar which generates the language.
- Every context-free grammar can be written in Chomsky Normal Form. Any non-empty string ω of length n generated by the grammar is done so using a derivation of $2n - 1$ steps.

We use these facts to decide a language of ordered pairs, with each pair containing a context-free grammar and a string it generates.

Theorem 4.7. *Let $A_{CFG} = \{\langle G, w\rangle : G$ is a context-free grammar that generates $w\}$. A_{CFG} is decidable.*

Proof. We construct a Turing Machine M as follows. On input $\langle G, w\rangle$, where G is a context-free grammar and w is a string, M begins by converting G to an equivalent grammar in Chomsky Normal Form. If $n > 0$, then M enumerates all derivations with $2n - 1$ steps, where $n = |w|$. Otherwise, M enumerates all derivations with a single step. M accepts $\langle G, w\rangle$ if and only if one such derivation generates w. Otherwise, M rejects $\langle G, w\rangle$. So M decides A_{CFG}. □

We next show that it is quite easy to decide if $L(G) = \emptyset$ for an arbitrary context-free grammar G. The proof of this next theorem utilizes a dynamic programming algorithm which is modeled after the algorithm to construct a Huffman encoding tree.

Theorem 4.8. *Let $E_{CFG} = \{\langle G\rangle : G$ is a context-free grammar and $L(G) = \emptyset\}$. E_{CFG} is decidable.*

Proof. We construct a Turing Machine M, which utilizes a dynamic programming procedure. We begin by mark each terminal symbol in the grammar G. For the recursive step, we mark a non-terminal symbol A if there exists a rule $A \to x_1x_2x_3 \ldots x_k$ where for each i, x_i was labeled at some previous iteration. The algorithm terminates if no additional symbol is marked at the last iteration. $L(G) \neq \emptyset$ if and only if S is marked (as tracing along the computation yields a derivation from S to some string of terminals). So M accepts G if and only if S is unmarked, and G rejects otherwise. □

We now show that every context-free language is decidable.

Theorem 4.9. *Let L be a context-free langauge. L is decidable.*

Proof. We construct a Turing Machine M to decide L as follows. Let S be the Turing Machine constructed in the proof of Theorem 4.7, and let G be a context-free grammar generating L. On input ω, M simulates S on $\langle G, \omega\rangle$ and accepts ω if and only if S accepts $\langle G, \omega\rangle$. M rejects ω otherwise. So M decides L, as S decides A_{CFG}. □

4.5 Undecidability

In the last section, we examined several problems which Turing Machines can decide, or solve. This section examines 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 algorithm 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.

Recall that both A_{DFA} and A_{CFG} were both decidable. The corresponding language for Turing Machines is given below:

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

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 the following result.

Theorem 4.10. *A language L is decidable if and only if L and $\overline{L}$ are recursively enumerable.*

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. □

In order to show that L_{diag} is undecidable, Theorem 4.10 provides 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 4.11. *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 4.12. *$\overline{L_{diag}}$ 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 $\omega_k \not\in \overline{L_{\text{diag}}}$. Then $\omega_k \not\in L(M_k) = \overline{L_{\text{diag}}}$. 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 $\omega_k \not\in \overline{L_{\text{diag}}}$. So $\overline{L_{\text{diag}}}$ is not recursively enumerable. □

Corollary 4.0.1. *L_{diag} is undecidable.*

Proof. This follows immediately from Theorem 4.10, as L_{diag} is recursively enumerable, while $\overline{L_{\text{diag}}}$ is not. □

4.6 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 125 (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 126 (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 4.13. *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. $\square$

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 4.14. *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. $\square$

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 4.15. *Let $E_{TM} = \{\langle M \rangle : M$ 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. $\square$

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 4.16. *Let $L_{ES} = \{\langle M \rangle : M$ 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. $\square$

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 4.17. *Let $L_{Reg} = \{L : L \text{ is regular }\}$. 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^n1^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 $L(M') = \{0^n1^n : n \in \mathbb{N}\}$ 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 4.18 (Rice)**.** *Let $\mathcal{R}$ be the set of recursively enumerable languages, and let C be a non-empty, proper subset of $\mathcal{R}$. 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 $\mathcal{R}$, $\overline{C}$ 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: 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.

5 Complexity Theory

The goal of Complexity Theory is to classify decidable problems according to the amount of resources required to solve them. Space and time are the two most common measures of complexity. Time complexity measures how many computations are required for a computer to solve decide an instance of the problem, with respect to the instance's size. Space complexity is analogously defined for the amount of extra space a computer needs to decide an instance of the problem, with respect to the instance's size.

In the previous sections, we have discussed various classes of computational machines- finite state automata, pushdown automata, and Turing Machines; as well as the classes of formal languages they accept. These automata answer decision problems: given a string $\omega \in \Sigma^*$, does ω belong to some language L? If L is regular, then a finite state automaton can answer this question. However, if L is only decidable, then the power of a Turing Machine (or Turing-equivalent model) is required. Formally, we define a decision problem as follows.

Definition 127 (Decision Problem)**.** Let Σ be an alphabet and let $L \subset \Sigma^*$. We say that the language L is a *decision problem.*

The complexity classes `P` and `NP` deal with decision problems. That is, they are sets of languages. In the context of an algorithms course, we abstract to the level of computational problems rather than formal languages. It is quite easy to formulate a computational decision problem as a language, though.

Example 134. Consider the problem of determining whether a graph G has a Hamiltonian cycle. The corresponding language would then be:

$$L_{HC} = \{\langle G\rangle : G \text{ is a graph that has a Hamiltonian Cycle}\}.$$

We would then ask if the string $\langle H\rangle$ is in L_{HC}. In other words, does H have a Hamiltonian Cycle? Note that $\langle H\rangle$ denotes an encoding of the graph H. That means the language L_{HC} contains string-representations of graphs, such that the graphs contain Hamiltonian Cycles. From a computational standpoint, we represent graph as finite data structures programatically. Examples include the adjacency matrix, the adjacency list, or the incidence matrix representations. As computers deal with strings, it is important that our mathematical objects be encoded as strings.

5.1 Time Complexity- $\texttt{P}$ and $\texttt{NP}$

Time complexity is perhaps the most familiar computational resource measure. We see this as early as our introductory data structures class. Nesting loops often result in $\mathcal{O}(n^2)$ runtime, and mergesort takes $\mathcal{O}(n \log(n))$ time to run. Junior and senior level data structures and algorithm analysis courses provide more rigorous frameworks to evaluate the runtime of an algorithm. This section does not focus on these tools. Rather, this section provides a framework to classify decision problems according to their time complexities. In order to classify such problems, it suffices to design a correct algorithm with the desired time complexity. This shows the problem is decidable in the given time complexity. With this in mind, we formalize the notion of time complexity.

Definition 128. Let $T : \mathbb{N} \to \mathbb{N}$ and let M be a Turing Machine that halts on all inputs. We say that M *has time complexity* $\mathcal{O}(T(n))$ if for every $n \in \mathbb{N}$, M halts in at most $T(n)$ steps on any input string of length n. We refer to:

$$\texttt{DTIME}(T(n)) = \{L \subset \Sigma^* : L \text{ is decided by some deterministic TM } M \text{ in time } \mathcal{O}(T(n))\}.$$

Remark: `DTIME` is the first complexity class we have defined. Observe that `DTIME` is defined based on a deterministic Turing Machine. Every complexity class must have some underlying model of computation. In order to measure complexity, it is essential that the computational machine is clearly defined. That is, we need to know what is running our computer program or algorithm to solve the problem. The formal language framework provides a notion of what the computer is reading. Intuitively, computers deal with binary strings. Programmers may work in an Assembly dialect, which varies amongst architectures, or some higher level language like Python or Java. In any case, the computer deals with some string representation of the algorithm as well as the problem.

With the definition of `DTIME` in tow, we have enough information to begin defining the class `P`.

Definition 129 (Complexity Class `P`)**.** The complexity class `P` is the set of languages that are decidable in polynomial time. Formally, we define:

$$\texttt{P} = \bigcup_{k \in \mathbb{N}} \texttt{DTIME}(n^k).$$

Example 135. The `Path` problem is defined as follows.

$$L_{\texttt{Path}} = \{\langle G, u, v\rangle : G \text{ is a graph; } u, v \in V(G), \text{ and } G \text{ has a path from } u \text{ to } v\}.$$

The `Path` problem is decidable in polynomial time using an algorithm like breadth-first search or depth-first search, both of which run in $\mathcal{O}(|V|^2)$ time. So $L_{\texttt{Path}} \in \texttt{P}$ since we have a polynomial time algorithm to decide $L_{\texttt{Path}}$.

Example 136. Relative primality is another problem that is decidable in polynomial time. We wish to check if two positive integers have no common positive factors greater than 1. Formally:

$$L_{\texttt{Coprime}} = \{(a, b) \in \mathbb{Z}^+ \times \mathbb{Z}^+ : \gcd(a, b) = 1\}.$$

We have $L_{\texttt{Coprime}} \in \texttt{P}$, as the Euclidean algorithm computes the gcd of two positive integers in $\mathcal{O}(\log(n))$ time, where $n = \max\{a, b\}$.

Remark: Note that the `Path` and `Coprime` problems are shown to be in `P` using conventional notions of an algorithm, which include random access. Turing Machines do not allow for random access, so it should raise an eyebrow about using more abstract notions of an algorithm to place problems into `P`. The reason we can do this is because the RAM model of computation is polynomial-time equivalent to the Turing Machine. That is, if we have a RAM computation that takes $T_1(n)$ steps on an input of size n, then a Turing Machine can simulate this RAM computation in $p_1(T_1(n))$ steps where p is some fixed polynomial. Similarly, if a Turing Machine executes a computation in $T_2(n)$ steps where n is the size of the input, then a RAM machine can simulate the Turing computation in $p_2(T(n))$ steps for some fixed polynomial $p_2(T(n))$. In fact, `P` can be equivocally defined using any model of computation that can simulate a Turing Machine in polynomial time.

Note that problems in `P` are considered computationally easy problems. Intuitively, computationally easy problems are those that can be solved in polynomial time. This does not mean that a problem is easy for which to develop a solution. Many of the algorithms to place problems in `P` are quite complex and elegant. For a long time, the `Linear Programming` problem was not known to be in `P`. The common algorithm was the Simplex procedure, which was developed in 1947 and is still taught in undergraduate and graduate optimization classes. In most cases, the Simplex algorithm works quite efficiently, but it does have degenerate cases resulting in exponential time computations. The Ellipsoid algorithm was developed in 1979, which placed the `Linear Programming` problem in `P`. In fact, `Linear Programming` is `P`-Complete, which means that it is one of the hardest problems in `P`. We will discuss what constitutes a complete problem later.

We now develop some definitions, which will allow us to define `NP`. Intuitively, the class `NP` contains problems for which correct solutions are easy to verify. The original definition of `NP` deal with non-deterministic Turing machines. Formally, the original definition is as follows.

Definition 130 (Complexity Class `NP` (Original Definition))**.** A language $L \in \texttt{NP}$ if there exists a non-deterministic Turing machine M and polynomial p such that for any $\omega \in L$, M accepts ω in $p(|\omega|)$ time.

This definition of `NP` has since been generalized to utilize verifiers. This generalized definition of `NP` actually implies the original definition of `NP`.

Definition 131 (Verifier)**.** A *verifier* for a language L is a Turing Machine M that halts on all inputs where:

$$L = \{\omega : M(\omega, c) = 1 \text{ for some string } c\}.$$

We refer to the string c as the *witness* or *certificate*. M is a *polynomial time verifier* if it runs in $p(|\omega|)$ time for a fixed polynomial p and every string $\omega \in L$. This implies that $|c| \leq p(|\omega|)$.

Definition 132 (Complexity Class `NP` (Modern Definition))**.** The complexity class `NP` is the set of languages that have polynomial time verifiers.

Intuitively, the class `NP` contains problems for which correct solutions are easy to verify. Intuitively, we have a Turing machine M, a string input ω, and the certificate c which provides a proof that ω belongs to L. The Turing Machine uses c to then verify $\omega \in L$. We can show both definitions of `NP` are equivalent.

Proposition 5.1. *A language L is decided by some non-deterministic Turing Machine M which halts in $p(|\omega|)$ steps on all inputs ω, for some fixed polynomial p, if and only if there exists some polynomial time verifier for L.*

Proof. Let L be a language and let p be a polynomial. Suppose first that L is decided by a non-deterministic Turing Machine M that halts on all inputs ω in $p(|\omega|)$ time steps. We construct a polynomial time verifier from M. Let $\hat{\delta_M}(\omega)$ be a complete accepting computation. Let M' be a verifier accepting strings in $\Sigma^* \times (Q(M))^*$. The verifier M' simulates M on ω visiting the sequence of states specified by $(Q(M))^*$. M' accepts $\langle \omega, \hat{\delta_M}(\omega) \rangle$ if and only if M accepts ω using the computation $\hat{\delta_M}(\omega)$. As M decides L in polynomial time, M' halts in polynomial time and $\hat{\delta_M}(\omega)$ is a certificate for ω. Thus, M' is a polynomial time verifier for L.

Conversely, let K be a polynomial time verifier for L. We construct a non-deterministic Turing Machine K' to accept L. On the input string ω, K' guesses a certificate C and simulates M on $\langle \omega, c \rangle$. K' accepts ω if and only if K accepts $\langle \omega, c \rangle$. Since K is a polynomial time verifier, K' will halt in polynomial time on all inputs and $\omega \in L(K')$ if and only if there exists some certificate c on which K accepts $\langle \omega, c \rangle$. So $L = L(K')$ and so K' is a non-deterministic Turing Machine accepting L. $\square$

Remark: The verifier definition of `NP` (see Definition 132) is of particular importance in areas such as interactive proofs and communication complexity.

We now develop some intuition about the class `NP`. In order to show a problem is in `NP`, we take an instance and provide a certificate, then construct a verifier. Just like with `P`, it suffices to provide a high level algorithm to verify an input and certificate pair due to the fact that our RAM computations are polynomial time equivalent to computations on a Turing machine.

Example 137. The `Hamiltonian Path` problem belongs to the class NP. Formally, we have:

$$L_{HP} = \{\langle G \rangle : G \text{ is a graph that has a Hamiltonian Path }\}.$$

Our instance is clearly $\langle G \rangle$, a string encoding of a graph. A viable certificate is a sequence of vertices that form a Hamiltonian path in G. We then check that the consecutive vertices in the sequence are adjacent, and that each vertex in the graph is included precisely once in the sequence. This algorithm takes $\mathcal{O}(|V|)$ time to verify the certificate, so $L_{HP} \in \texttt{NP}$.

Example 138. Deciding if a positive integer is composite is also in NP. Recall that a composite integer $n > 1$ can be written as $n = ab$ where $a, b \in \mathbb{Z}^+$ and $1 < a, b < n$. That is, n is not prime. Formally:

$$L_{\texttt{Composite}} = \{n \in \mathbb{Z}^+ : \exists a, b \in [n-1] \text{ s.t. } n = ab\}.$$

Our instance is $n \in \mathbb{Z}^+$ and a viable certificate is a sequence of positive integers $a_1, \ldots, a_k$ such that each $a_i \in [n-1]$ and $\prod_{i=1}^{k} a_i = n$. It takes $\mathcal{O}(k)$ time to verify that the certificate is a valid factorization of n. As $k < n$, we have a clear polynomial time algorithm to verify an integer is composite given a certificate.

We now arrive at the P = NP problem. Intuitively, the P = NP problem asks if every decision problem that can be easily verified can also be easily solved. It is straight-forward to show that P ⊂ NP. It remains open as to whether NP ⊂ P.

Proposition 5.2. *P ⊂ NP.*

Proof. Let $L \in \texttt{P}$ and let M be a deterministic, polynomial time Turing Machine that decides L. We construct a polynomial time verifier M' for L as follows. Let $\omega \in \Sigma^*$. On input $\langle \omega, 0 \rangle$, M' simulates M on ω ignoring the certificate. M' accepts $\langle \omega, 0 \rangle$ if and only if M accepts ω. Since M decides L in polynomial time, M' is thus a polynomial time verifier for L. So $L \in \texttt{NP}$. □

5.2 NP-Completeness

The P = NP problem has been introduced at a superficial level- are problems whose solutions can be verified easily also easy to solve? In some cases, the answer is yes- for the problems in P. In general, this is unknown. However, it is widely believed that P ≠ NP. In this section, the notion of NP-Completeness will be introduced. NP-Complete problems are the hardest problems in NP and are widely believed to be intractible. We begin with the notion of a reduction.

Definition 133 (Polynomial Time Computable Function)**.** A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time TM M exists that halts with just $f(w)$ on the tape when started on w.

Definition 134 (Polynomial Time Reducible)**.** Let $A, B \subset \Sigma^*$. We say that A is *polynomial time reducible* to B, which is denoted $A \leq_p B$ if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $\omega \in A$ if and only if $f(\omega) \in B$.

The notion of reducibility provides a partial order on computational problems with respect to hardness. That is, suppose A and B are problems such that $A \leq_p B$. Then an algorithm to solve B can be used to solve A. Suppose we have the corresponding polynomial time reduction $f : \Sigma^* \to \Sigma^*$ to reduce A to B. Formally, we take an input $\omega \in \Sigma^*$ and transform it into $f(\omega)$. We use a decider for B to decide if $f(\omega) \in B$. As $\omega \in A$ if and only if $f(\omega) \in B$, we have an algorithm to decide if $\omega \in A$. This brings us to the definition of NP-Hard.

Definition 135 (NP-Hard)**.** A problem A is NP-Hard if for every $L \in \texttt{NP}$, $L \leq_p A$.

Definition 136 (NP-Complete)**.** A language L is NP-Complete if $L \in \texttt{NP}$ and L is NP-Hard.

Remark: Obseve that every NP-Complete problem is a decision problem. In general, NP-Hard problems need not be decision problems. Optimization and enumeration problems are common examples of NP-Hard problems. Note as well that any two NP-Complete languages are polynomial time reducible to each other, and so are equally hard. That is, a solution to one NP-Complete language is a solution to all NP-Complete languages. This leads to the following result.

Theorem 5.1. *Let L be an NP-Complete language. If $L \in$ P, then P = NP.*

Proof. Proposition 5.2 already provides that $\mathtt{P} \subset \mathtt{NP}$. So it suffices to show that $\mathtt{NP} \subset \mathtt{P}$. Let $L \in \mathtt{NP}$ and let K be an NP-Complete language that is also in P. Let $f : \Sigma^* \to \Sigma^*$ be a polynomial time reduction from L to K, and let M be a polynomial time decider for K. Let $\omega \in L$. We transform ω into $f(\omega)$ and run M on $f(\omega)$. From the reduction, we have $\omega \in L$ if and only if $f(w) \in K$ accepts $f(\omega)$. Since M is a decider, we have a polynomial time decider for L. Thus, $L \in \mathtt{P}$ and we have $\mathtt{P} = \mathtt{NP}$. □

In order to show that a language L is NP-Complete, it must be shown that $L \in \mathtt{NP}$ and for every language $K \in \mathtt{NP}$, $K \leq L$. Constructing a polynomial-time reductions from each language in NP to the target language L is not easy. However, if we already have an NP-Complete problem J, it suffices to show $J \leq_p L$, which shows L is NP-Hard. Of course, in order to use this technique, it is necessary to have an NP-Complete language with which to begin. The Cook-Levin Theorem provides a nice starting point with the Boolean Satisfiability problem, better known as SAT. There are several variations on the Cook-Levin Theorem. One variation restricts to CNF-SAT, in which the Boolean formulas are in *Conjunctive Normal Form*. Another version shows that the problem of deiding if a combinatorial circuit is satisfiable, better known as `Circuit SAT`, is NP-Complete. We begin with a some definitions.

Definition 137 (Boolean Satisfiability Problem (SAT))**.**

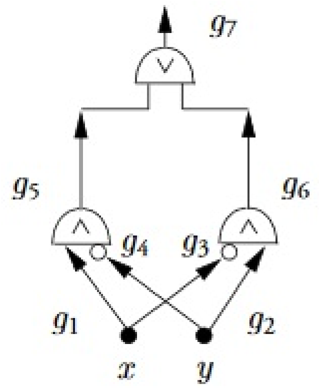
- `Instance`: Let $\phi : \{0,1\}^n \to \{0,1\}$ be a Boolean function, restricted to the operations of AND, OR, and NOT.
- `Decision`: Does there exist an input vector $x \in \{0,1\}^n$ such that $\phi(x) = 1$?

Example 139. The Boolean function $\phi(x_1, x_2, x_3) = x_1 \vee x_2 \wedge \overline{x_3}$ is an instance of SAT.

We next introduce the combinatorial circuit.

Definition 138 (Combinatorial Circuit)**.** A *Combinatorial Circuit* is a directed acyclic graph where the vertices are labeled with a Boolean operation or variable (input). Each operation computes a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, with the vertex having n incoming arcs, and m outgoing arcs.

Example 140. Consider the following combinatorial circuit. Here, x and y are the input vertices which feed into AND gates and NOT gates. The white vertices are the NOT gates, and the vertices labeled $\wedge$ are the AND gates. Each AND gate then feeds into an OR gate, which produces the final output of 0 or 1.



With this in mind, we define `Circuit SAT`, our first NP-Complete language.

Definition 139 (`Circuit SAT`)**.**

- `Instance:` Let C be a combinatorial circuit with a single output.
- `Decision:` Does there exist an input vector $x \in \{0,1\}^n$ such that $C(x) = 1$?

Theorem 5.2 (Cook-Levin)**.** *`Circuit SAT` is NP-Complete.*

The proof of the Cook-Levin Theorem is quite involved. We sketch the ideas here. In order to show `Circuit SAT` is in NP, it is shown that a Turing Machine can simulate a combinatorial circuit taking T steps in $p(T)$ steps for a fixed polynomial p. This enables a verifier to be constructed for a given combinatorial circuit. In order to show that `Circuit SAT` is NP-Hard, it is shown that any Turing Machine can be simulated by a combinatorial circuit with a polynomial time transformation. Since NP is the set of languages with polynomial time verifiers, this shows that each verifier can be transformed into a combinatorial circuit with a polynomial time computation. So `Circuit SAT` is NP-Complete.

With `Circuit SAT` in tow, we can begin proving other languages are NP-Complete, starting with CNF-SAT which we introduce below. Note that we could have started with any NP-Complete problem. `Circuit SAT` happened to be proven NP-Complete without an explicit reduction from another NP-Complete problem; hence out choice of it.

Definition 140 (Clause)**.** A *Clause* is a Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ where ϕ is written as consists of variables or their negations, all added together (where addition is the OR operation).

Definition 141 (Conjunctive Normal Form)**.** A Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ is in *Conjunctive Normal Form* if $\phi = C_1 \wedge C_2 \wedge \ldots C_k$, where each C_i is a clause.

Definition 142 (`CNF-SAT`)**.**

- `Instance:` A Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ in Conjunctive Normal Form.
- `Decision:` Does there exist an input vector $x \in \{0,1\}^n$ such that $\phi(x) = 1$?

Example 141. The Boolean function $\phi(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3})$ is in Conjunctive Normal Form.

Theorem 5.3. *CNF-SAT is NP-Complete.*

Proof. In order to show `CNF-SAT` is NP-Complete, we show that `CNF-SAT` is in NP and `CNF-SAT` is NP-Hard.

- **Claim 1:** `CNF-SAT` is in NP.

  Proof. In order to show `CNF-SAT` is in NP, it suffices to exhibit a polynomial time verification algorithm. Let $\phi : \{0,1\}^n \to \{0,1\}$ be a Boolean function with k literals (either a variable or its negation). Let $x \in \{0,1\}^n$ such that $\phi(x) = 1$. We simply evaluate $\phi(x)$, which takes $O(k)$ time. So `CNF-SAT` is in NP. □

- **Claim 2:** CNF-SAT is NP-Hard.

  Proof. We show `Circuit SAT` $\leq_p$ `CNF-SAT`. Let C be a combinatorial circuit. We convert C to a Boolean function as follows. For each vertex v of C, we construct a Boolean function ϕ in Conjunctive Normal Form as follows.

  – If v is an input for C, then construct the literal x_v.
  – If v is the NOT operation with input x_k, we create the variable x_v and include the following clauses in ϕ: $(x_v \vee x_k)$ and $(\overline{x_v} \vee \overline{x_k})$. Thus, in order for ϕ to be satisfiable, it is necessary that $x_v = \overline{x_k}$.
  – If v is the OR operation with inputs x_i, x_j, we create the variable x_v and include the following clauses in ϕ: $(x_v \vee \overline{x_i}), (x_v \vee \overline{x_j}), (\overline{x_i} \vee x_i \vee x_j)$. Thus, in order for ϕ to be satisfiable, it is necessary that $x_v = 0$ if and only if $x_i = x_j = 0$, which realizes the OR operation from C.
  – If v is the AND operation with inputs x_i, x_j, we create the variable x_v and include the following clauses in ϕ: $(\overline{x_v} \vee x_i), (\overline{x_v} \vee x_j), (x_v \vee \overline{x_i} \vee \overline{x_j})$. Thus, in order for ϕ to be satisfiable, it is necessary that $x_v = 1$ if and only if $x_i = x_j = 1$, which realizes the AND operation from C.
  – If v is the output vertex, we construct the variable x_v and add it to ϕ.

  There are at most $9|V|$ literals in ϕ, where $|V|$ is the number of vertices in C. So this construction occurs in polynomial time. It suffices to show that C is satisfiable if and only if ϕ is satisfiable. Suppose first C is satisfiable. Let $x \in \{0,1\}^n$ be a satisfying configuration for C. For each logic gate vertex v, set x_v to be the resultant of that operation on the inputs. By the analysis during the construction of ϕ, we have that ϕ is satisfiable. Conversely, suppose ϕ is satisfiable with input configuration $y \in \{0,1\}^k$ where k is the number of clauses. The first n elements of y, $(y_1, \ldots, y_n)$ corresponding to the input vertices of C form a satisfying configuration for C. So `CNF-SAT` is NP-Hard. □

□

We now reduce `CNF-SAT` to the general `SAT` problem to show `SAT` is `NP`-Complete.

Theorem 5.4. *`SAT` is `NP`-Complete.*

Proof. The procedure to show `CNF-SAT` $\in$ `NP` did not rely on the fact that the Boolean functions were in Conjunctive Normal Form. So this same procedure also suffices to show `SAT` is in `NP`. As `CNF-SAT` is a subset of `SAT`, the inclusion map f : `CNF-SAT` $\to$ `SAT` sending $f(\phi) = \phi$ is a polynomial time reduction from `CNF-SAT` to `SAT`. So `SAT` is `NP`-Hard. Thus, `SAT` is `NP`-Complete. □

From `CNF-SAT`, there is an easy reduction to the `Clique` problem.

Definition 143 (`Clique`)**.**

- `Instance:` Let G be a graph and $k \in \mathbb{N}$.
- `Decision:` Does G contain a complete subgraph with k vertices?

Theorem 5.5. *`Clique` is `NP`-Complete.*

Proof. We show that `Clique` is in `NP` and that `Clique` is `NP`-Hard.

- **Claim 1:** `Clique` is in `NP`.

  Proof. Let (G, k) be an instance of `Clique`. Let $S \subset V(G)$ be a set of vertices that induce a complete subgraph on k vertices. We check that all $\binom{k}{2}$ edges are present in $G[S]$, the subgraph of G induced by S. This takes $\mathcal{O}(n^2)$ time, which is polynomial. So `Clique` is in `NP`. □

- **Claim 2:** `Clique` is `NP`-Hard.

  Proof. It suffices to show `CNF-SAT` $\leq_p$ `Clique`. Let ϕ be an instance of `CNF-SAT` with k-clauses. We construct an instance of `Clique` as follows. Let G be the graph we construct. Each occurrence of a variable in ϕ corresponds to a vertex in G. We add all possible edges except if: (1) two vertices belong to the same clause; or (2) if two vertices are contradictory. If the length of ϕ is n, then this construction takes $\mathcal{O}(n^2)$ time which is polynomial time. It suffices to show that ϕ is satisfiable if and only if there exists a k-clique in G.

  Suppose first ϕ is satisfiable. Let x be a satisfying configuration for ϕ. As ϕ is in Conjunctive Normal Form, there exists a literal in each clause that evaluates to 1. We select one such literal from each clause. As none of these literals are contradictory, the corresponding vertices in G induce a k-`Clique`.

  Conversely, suppose G has a k-clique. Let $S \subset V(G)$ be a set of vertices that induce a k-`Clique` in G. If $v \in S$ corresponds to a variable x_i, then we set $x_i = 1$. Otherwise, v corresponds to a variable's negation and we set $x_i = 0$. Any variable not corresponding to a vertex in the set is set to 0. Recall that each vertex in S corresponds to a literal from each clause and the literals are not contradictory. As ϕ is in Conjunctive Normal Form, we have a satisfying configuration for ϕ. We conclude that `Clique` is `NP`-Hard. □

□

The `Clique` problem gives us two additional `NP`-Complete problems. The first problem is the `Independent Set` problem, and the second is the `Subgraph Isomorphism` problem. The `Subgraph Isomorphism` problem is formally:

Definition 144 (`Subgraph Isomorphism`)**.**

$$L_{SI} = \{\langle G, H\rangle : G, H \text{ are graphs, and } H \subset G\}.$$

The inclusion map from `Clique` to `Subgraph Isomorphism` provides that `Subgraph Isomorphism` is `NP`-Hard. It is quite easy to verify that H is a subgraph of G given an isomorphism.

An independent set is the complement of a `Clique`. Formally, we have the following.

Definition 145 (Independent Set)**.** Let G be a graph. An independent set is a set $S \subset V(G)$ such that for any $i, j \in S$, $ij \notin E(G)$. That is, all vertices of S are pairwise non-adjacent in G.

This leads to the `Independent Set` problem.

Definition 146 (`Independent Set` (Problem))**.**

- `Instance:` Let G be a graph and $k \in \mathbb{N}$.
- `Decision`: Does G have an independent set with k vertices?

Theorem 5.6. *`Independent Set` is `NP`-Complete.*

Proof. We show that `Independent Set` is in `NP`, and that `Independent Set` is `NP`-Hard.

- **Claim 1:** `Independent Set` is in `NP`.

  Proof. Let $\langle G, k\rangle$ be an instance of `Independent Set` and let $S \subset V(G)$ be an independent set of order k. S serves as our certificate. We check that for each distinct $i, j \in S$, $ij \notin E(G)$. This check takes $\binom{k}{2}$ steps, which is $\mathcal{O}(|V|^2)$ time. So `Independent Set` is in `NP`. □

- **Claim 2:** `Independent Set` is `NP`-Hard.

  Proof. We show `Clique` $\leq_p$ `Independent Set`. Let $\langle G, k\rangle$ be an instance of `Clique`. Let $\overline{G}$ be the complement of G, in which $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \{ij : i, j \in V(G), ij \notin E(G)\}$. This construction takes $\mathcal{O}(|V|^2)$ time. So this construction is in polynomial time. As an independent set is the complement of a `Clique`, G has a k-`Clique` if and only if $\overline{G}$ has an independent set with k-vertices. So `Independent Set` is `NP`-Hard. □

□

We provide one more `NP`-Hardness proof to illustrate that not all `NP`-Hard problems are in `NP`. We introduce the `Hamiltonian Cycle` and `TSP-OPT` problems.

Definition 147. `Hamiltonian Cycle`

- `Instance:` Let $G(V, E)$ be a graph.
- `Decision:` Does G contain a cycle visiting every vertex in G?

And the Traveling Salesman optimization problem is defined as follows.

Definition 148. `TSP-OPT`

- `Instance:` Let $G(V, E, W)$ be a weighted graph where $W : E \to \mathbb{R}_+$ is the weight function.
- `Solution:` Find the minimum cost Hamiltonian Cycle in G.

We first note that `Hamiltonian Cycle` is `NP`-Complete, though we won't prove this. In order to show `TSP-OPT` is `NP`-Hard, we reduce from `Hamiltonian Cycle`.

Theorem 5.7. *`TSP-OPT` is `NP`-Hard.*

Proof. We show `Hamiltonian Cycle` $\leq_p$ `TSP-OPT`. Let G be a graph with n vertices and a Hamiltonian cycle C. We construct a weighted K_n as follows. Each edge in K_n corresponding to C is weighted 0. All other edges are weighted 1. So any minimum weight Hamiltonian cycle in K_n has weight at least 0. We show that G has a Hamiltonian cycle if and only if the minimum weight Hamiltonian cycle in the K_n has weight 0. Suppose first G has a Hamiltonian cycle C. We trace along C in K_n to obtain a Hamiltonian cycle of weight 0 in K_n. Conversely, suppose K_n has a Hamiltonian cycle of weight 0. By construction, this corresponds to the Hamiltonian cycle C in G. We conclude that `Hamiltonian Cycle` $\leq_p$ `TSP-OPT`, so `TSP-OPT` is `NP`-Hard. □

5.3 More on P and P-Completeness

In this section, we explore the complexity class P as well as P-Completeness. Aside from containing languages that are decidable in polynomial time, P is important with respect to parallel computation. Just as NP-Hard problems are difficult to solve using a sequential model of computation, P-Hard problems are difficult to solve in parallel. We omit exposition on parallel computation. Rather, there are two big takeaways. The first is that many NP-Complete languages have subsets which are easily decidable. The second important takeaway is a clear understanding of P-Completeness.

We begin with the 2-SAT problem.

Definition 149 (k-`CNF-SAT`)**.**

- `Instance:` A Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ in Conjunctive Normal Form, where each clause has precisely k literals.
- `Decision:` Does there exist an input vector $x \in \{0,1\}^n$ such that $\phi(x) = 1$?

It is a well known fact that k-`CNF-SAT` is NP-Complete for every $k \geq 3$. However, 2-`CNF-SAT` is actually in P. One proof of this is by a reduction to another problem in P: the `Strongly Connected Component` problem. We can decide the `Strongly Connected Component` problem using Tarjan's Algorithm, an $\mathcal{O}(|V| + |E|)$ time algorithm. We define the `Strongly Connected Component` problem formally.

Definition 150 (`Strongly Connected Component` (`SCC`))**.**

- `Instance:` A directed graph $G(V, E)$.
- `Decision:` Do there exist vertices i, j such that there are directed $i \to j$ and $j \to i$ paths in G?

Theorem 5.8. *2-CNF-SAT is in* P.

Proof. We show 2-`CNF-SAT` $\leq_p$ `SCC`. We begin by constructing the implication graph G, which is a directed graph. The vertices of G are the components of x and their negations, yielding $2n$ vertices in total. For each clause in C $(x_i \vee x_j)$, add directed edges $(\neg x_i, x_j), (\neg x_j, x_i)$. (So for example, if a clause contained $(\neg x_2 \vee x_3)$, the edges added to G would be $(x_2, x_3), (\neg x_3, \neg x_2)$.) The reduction to `SCC` looks at the implication graph to determine if there is a component x_i such that there is a directed path x_i to $\neg x_i$, and a directed path $\neg x_i tox_i$. Notice that there are at most $\binom{n}{2}$ clauses to examine and so at most $\binom{2n}{2}$ edges to add to G, so the reduction is polynomial in time.

So we need to prove a couple facts.

- If there exists an $a \to b$ directed path in G, then there exists a directed $\neg b \to \neg a$ path in G. We will use this fact to justify the existence of a strongly connected component if there is a directed $x_i \to \neg x_i$.
- C is satisfiable if and only if there is no strongly connected component in G containing both a variable and its negation. This will substantiate the validity of the reduction.

We proceed with proving these claims:

- **Claim 1**: If there exists a directed $a \to b$ path in G, then there exists a directed $\neg b \to \neg a$ path in G.

  Proof. Suppose there exists an $a \to b$ directed path in G. By construction, for each edge $(c, d) \in G$, there exists an edge (d, c). So given the $a \to b$ directed path: $a \to p_1 \to ... \to p_k \to b$, there exist directed edges in G: $(\neg b, \neg p_k), ..., (\neg p_1, \neg a)$, yielding a directed $\neg b \to \neg a$ path, as claimed. □

- **Claim 2**: C is satisfiable if and only if there is no strongly connected component in G.

Proof. It will first be shown that if C is satisfiable, then there is no strongly connected component in G. This will be done by contradiction. Let x be a satisfying configuration of C, and let x_i such that there is a strongly connected component including x_i and $\neg x_i$. Let $x_i \to p_1 \to ... \to p_n \to \neg x_i$ be the directed $x_i \to \neg x_i$ path in G. Suppose first $x_i = 1$. By construction, the edges $(\neg x_i, p_1), (\neg p_i, p_{i+1})$ for each $i = 1, ..., n-1$; and $(\neg p_n, \neg x_n)$ are in G. And so for each $i = 1, ..., n$, p_i must be 1 to satisfy the corresponding clause in C. However, since $\neg x_i$ is 0, p_n must be 0 to satisfy $(\neg p_n, \neg x_n)$, a contradiction. By similar analysis, x_i cannot be 0 either. And so C is unsatisfiable. Thus, if C is satisfiable, there is no strongly connected component containing both x_i and $\neg x_i$.

Now suppose there is no strongly connected component in G. It will be shown that C is satisfiable by contradiction. Suppose there are no $x_i \in x$ such that there exists a directed $x_i \to \neg x_i$ path. For each unmarked vertex $v \in V(G)$ such that no $v \to \neg v$ path exists, mark v as 1. Now mark each neighbor of v as 1, and the negations of each marked variable as 0. Repeat this process until all vertices have been marked. By finiteness of the graph, this process terminates. Since there are no strongly connected components in G, all vertices will be marked. As C is unsatisfiable, let $i, j \in \{1, ..., n\}$ such $i \neq j$ and that there exists directed $x_i \to x_j$ and $x_i \to \neg x_j$ paths. So x_i implies both x_j and $\neg x_j$, which is a fallacy. By construction of G, there must exist a $\neg x_j \to \neg x_i$ directed path in G, which implies that G has a strongly connected component. However, G has no strongly connected component by assumption, a contradiction.

Thus, we conclude C is satisfiable if and only if there exists no strongly connected component in G, proving Claim 2. □

As 2-`CNF-SAT` $\leq_p$ `SCC`, it follows that 2-`CNF-SAT` is in `P`. □

Another example of an `NP`-Complete problem that has a subset in `P` is the `Hamiltonian Cycle` problem. Consider the subset $\{\langle C_n \rangle : n \geq 3\}$. It is easy to check if a graph is a cycle; and hence, has a Hamiltonian cycle.

We now introduce the notion of a `P`-Hard problem. A `P`-Hard problem is defined similarly as an `NP`-Hard problem, with the exception of the fact that the reductions are bounded in space rather than time. Formally, the reductions have to be computable with an additional logarithmic amount of space based on the input string. In fact, a log-space computation is necessarily polynomial time. We will prove this when we discuss the complexity class `PSPACE` and space complexity.

Definition 151 (Log-Space Computable Function)**.** A function $f : \Sigma^* \to \Sigma^*$ is a *log-space computable function* if some TM M exists that halts with just $f(w)$ on the tape when started on w, and uses at most $\mathcal{O}(\log(|\omega|))$ additional space.

Definition 152 (Log-Space Reducible)**.** Let A, B be languages. We say that A is *log-space reducible* to B, denoted $A \leq_\ell B$, if there exists a log-space computable function $f : \Sigma^* \to \Sigma^*$ such that $\omega \in A$ if and only if $f(\omega) \in B$.

Definition 153 (`P`-Hard)**.** A problem K is `P`-Hard if for every $L \in$ `P`, $L \leq_\ell K$.

And so we now define `P`-Complete analogously to `NP`-Complete.

Definition 154 (`P`-Complete)**.** A language L is `P`-Complete if $L \in$ `P` and L is `P`-Hard.

The proof of the Cook-Levin Theorem provides us a first `P`-Complete problem: `Circuit Value`. The `Circuit SAT` problem takes a combinatorial circuit and asks if it is satisfiable. The `Circuit Value` problem takes a combinatorial circuit and an input vector as the instance, and the decision problem is if the input vector satisfies the circuit.

Definition 155 (`Circuit Value` (CV))**.**

- `Instance`: Let C be a combinatorial circuit computing a function $\phi : \{0,1\}^n \to \{0,1\}$, and let $x \in \{0,1\}^n$.
- `Decision`: Is $C(x) = 1$?

Theorem 5.9. *`Circuit Value` is P-Complete.*

With `Circuit Value` in mind, we prove another P-Complete problem: `Monotone Circuit Value`. The difference between `Circuit Value` and `Monotone Circuit Value` is that we restrict to the operations of {AND, OR} in `Monotone Circuit Value`. So in order to prove `Monotone Circuit Value`, we flush the negations down to the inputs using DeMorgan's Law. We define `Monotone Circuit Value` formally below.

Definition 156 (`Monotone Circuit Value (MCV)`)**.**

- `Instance`: Let C be a combinatorial circuit in which only AND and OR gates are used, and let $\phi : \{0,1\}^n \to \{0,1\}$ be the function C computes. Let $x \in \{0,1\}^n$.
- `Decision`: Is $\phi(x) = 1$?

In order to prove `MCV` is `P`-Complete, we need a few important facts:

- All Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$ can be computed using the operations `And`, `Or`, and `Not`.
- All Boolean functions can be computed using operations equivalent to `And`, `Or`, and `Not`.
- All logical circuits can be written as straight-line programs. That is, we have only variable assignments and arithmetic being performed. There are no loops, conditionals, or control structures of any kind.

We begin by sketching the proof that `MCV` is `P`-Complete, so the ideas are clear. Since `MCV` is a subset of CV (we take circuits without the NOT operation, which are also instances of CV), `MCV` is in `P`. To show `MCV` is `P`-Hard, we show CV $\leq_\ell$ `MCV`. That is, for each instance of `CV`, a corresponding instance of `MCV` will be constructed. This is difficult though, as it is necessary to get rid of the `Not` operations from `CV`. We do this by flushing the `Not` operations down each layer of the circuit using DeMorgan's Law.

We then construct Dual-Rail Logic (DRL) circuits, where each variable x_i from x in the `CV` instance is represented as $(x_i, \neg x_i)$ in the `MCV` instance. So any negations we may want are constructed up-front, so the `NOT` operation becomes unnecessary. The DRL operations are defined as follows:

- `DRL-And`: $(x, \neg x) \wedge (y, \neg y) = (x \wedge y, \neg(x \wedge y)) = (x \wedge y, \neg x \vee \neg y)$
- `DRL-Or`: $(x, \neg x) \vee (y, \neg y) = (x \vee y, \neg(x \vee y)) = (x \vee y, \neg x \wedge \neg y)$
- `DRL-Not`: $\neg(x, \neg x)$ is given just by twisting the wires, sending x and $\neg x$ in opposite directions.

Since the `Not` operation is given upfront in the variable declarations, the DRL operations are all realizable over the monotone basis of $\{\texttt{And}, \texttt{Or}\}$. DRL is also equally as powerful as the basis $\{\texttt{And}, \texttt{Or}, \texttt{Not}\}$. So any Boolean function can be computed with DRL Circuits.

We now formally prove `MCV` is `P`-Complete.

Theorem 5.10. *`Monotone Circuit Value` is P-Complete.*

Proof. In order to show `MCV` is `P`-Complete, we show that `MCV` is in `P` and every problem in `P` is log-space reducible to `MCV` (ie., `MCV` is `P`-Hard). As `MCV` is a subset of CV and CV is in `P`, it follows that `MCV` is in `P`.

To show `MCV` is `P`-Hard, we show `CV` $\leq_\ell$ `MCV`. Let (C, x) be an instance of `CV` where C is the circuit over the basis $\{\texttt{And}, \texttt{Or}, \texttt{Not}\}$ and x is the input sequence. We construct C' over the monotone basis $\{\texttt{And}, \texttt{Or}\}$ from C, by rewriting C as a dual-rail circuit. Let $P(C)$ be the straight-line program representing C. Let P' be the straight-line program used to construct C'. For each line n in $P(C)$, let this instruction be line $2n$ in P'. Line $2n + 1$ in P' corresponds to the negation of line n in $P(C)$.

The `Not` operation in $P(C)$ is realized in P' by twisting the wires. That is, the step $(2k = \neg 2i)$ is realized by the steps $(2k = 2i + 1)$ and $(2k + 1 = 2i)$. The `And` operation in $P(C)$ $(2k = 2i \wedge 2j)$ is replaced by the steps $(2k = 2i \wedge 2j)$ and $(2k + 1 = (2i + 1) \vee (2j + 1))$. Finally, the `Or` operation $(2k = 2i \vee 2j)$ is realized by $(2k = 2i \vee 2j)$ and $(2k + 1 = (2i + 1) \wedge (2j + 1))$. And so $P(C) = P'$ for all inputs. So $P(C) = 1$ if and only if $P' = 1$, and $P(C) = 0$ if and only if $P' = 0$. So the reduction is valid.

It now suffices to argue the reduction takes a logarithmic amount of space. Generating P' from $P(C)$ can be done using a counter variable. So for each step i in $P(C)$, we perform operations at lines $2i$ and $2i+1$ in P'. So if there are n steps in $P(C)$, we need $\log_2(\lceil 2n+1 \rceil)$ bits, which grows asymptotically with $c(\log_2(2)+\log_2(n)) = c(1+\log_2(n))$ for some integer constant $c > 1$. So the amount of space required is $\mathcal{O}(log(n))$. And so we conclude that MCV is P-Complete. □

5.4 Closure Properties of NP and P

TODO

5.5 Structural Proofs for NP and P

TODO

5.6 Ladner's Theorem

The weak version of Ladner's Theorem states that if $\texttt{P} \neq \texttt{NP}$, then there exists a problem $L \in \texttt{NP} \setminus \texttt{P}$ such that $L \not\in \texttt{NP}$-Complete. We refer to the set $\texttt{NP} \setminus \texttt{P}$ as NP-Intermediate. The consequence of Ladner's Theorem is that finding an NP-Intermediate language would settle the $\texttt{P} = \texttt{NP}$ problem, providing a separation. Ladner's Theorem can be strengthened to provide an infinite strict heirarchy of NP-Intermediate languages. In this section, we provide Ladner's original proof of the weak Ladner Theorem, as well as the stronger version of Ladner's Theorem. Additionally, we provide Russell Impagliazzo's proof of the weak version of Ladner's Theorem.

Ladner proved the weak version of Ladner's theorem as follows. Define the language:

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

where f is a function we will construct later. Here, L is our target NP-Intermediate language. The goal is to "blow holes" in L, so that L is not NP-Complete, while also ensuring L is not "easy enough" to be in P. We accomplish this by diagonalizing against polynomial time reductions, as well as polynomial time deciders. The trick is to ensure that f is computable in polynomial time, which ensures that $L \in \texttt{NP}$.

In order to accomplish this, f tracks the given stage. At even-indexed stages (i.e., $f(n) = 2i$), we diagonalize against the ith polynomial time decider. While at odd-indexed stages (i.e., $f(n) = 2i+1$), we diagonalize against polynomial time reductions from SAT to L. That is, we want that $\texttt{SAT} \not\leq_p L$. As SAT is NP-Complete, this ensures that L is *not* NP-Complete.

We now proceed with the formal proof.

Theorem 5.11 (Ladner (Weak), 1975). *If $\texttt{P} \neq \texttt{NP}$, then there exists a language $L \in \texttt{NP} \setminus \texttt{P}$, such that $L \not\in \texttt{NP}$-Complete.*

Proof. Define:

$$L := \{x \in \texttt{SAT} : f(|x|) \text{ is even } \}.$$

Let $(M_i)_{i \in \mathbb{Z}^+}$ be an enumeration of polynomial-time Turing machines, which enumerates the languages in 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 reductions to SAT.

Now let $M_{\texttt{SAT}}$ be a decider for SAT. 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 it. On input 1^n (with $n > 1$), M_f proceeds in two stages, each lasting exactly n steps. During the first stage, M_f computes $f(0), f(1), \ldots,$ until it runs out of time. Suppose the last value M_f computed at the first stage was $f(x) = k$. At the next stage, the output of M_f will either be k or $k+1$, to be determined in the second stage.

In 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 $\texttt{P}$ as follows. The goal is to find a string $z \in \Sigma^*$ such that $z \in (L(M_i) \triangle L)$. We enumerate such strings z in lexicographic order, and then computing $M_i(z), M_{\texttt{SAT}}(z)$, and $f(|z|)$ for all such strings. Note that by definition of L, we must compute $f(|z|)$ to ensure that $f(|z|)$ is even. If such a string z 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 and need to do so on the next iteration).

- **Case 2:** Suppose that $k = 2i - 1$. Here, we diagonalize against polynomial-time computable functions. In this case, M_f searches for a string $z \in \Sigma^*$ such that F_i is an incorrect Karp reduction on z. That is, either:

  - $z \in \texttt{SAT}$ and $F_i(z) \not\in L$; or
  - $z \not\in \texttt{SAT}$ and $F_i(z) \in L$.

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

Claim: $L \not\in \texttt{P}$.

Proof. Suppose to the contrary that $L \in \texttt{P}$. Let M_i be a TM that decides L. By Case 1 in the second stage of the construction of M_f, no string z is found satisfying $z \in L$ and $z \not\in L(M_i)$. Thus, $f(n)$ is even for all but finitely many n. Thus, L and $\texttt{SAT}$ coincide for all but finitely many strings. It follows that $\texttt{SAT}$ is decidable in polynomial time (decide if a string is in L; if not, we only have finitely many cases to check). So $\texttt{SAT} \in \texttt{P}$, contradicting the assumption that $\texttt{P} \neq \texttt{NP}$. □

Claim: $L \not\in \texttt{NP}$-Complete.

Proof. Suppose to the contrary that L is $\texttt{NP}$-Complete. Then there is a polynomial time reduction F_i from $\texttt{SAT}$ to L. So $f(n)$ will be even for only finitely many n, which implies that L is finite. So $L \in \texttt{P}$, which implies that $\texttt{SAT} \in \texttt{P}$, contradicting the assumption that $\texttt{P} \neq \texttt{NP}$. □

□

Theorem 5.14, the weak Ladner's Theorem, can be strengthened to provide an infinite strict heirarchy of $\texttt{NP}$-Intermediate languages. The proof of this stronger version of Ladner's Theorem is almost identical to the proof of the weak version, Theorem 5.14. Given an $\texttt{NP}$-Intermediate langauge L_i, we construct L_{i+1} by diagonalizing against polynomial time Turing Machines to ensure that $L_{i+1} \not\in \texttt{P}$. We also diagonalize against polynomial time reductions from L_i, to L_{i+1}. In order to ensure we have a heirarchy, we also need that $L_{i+1} \leq_p L_i$. Blowing holes in L_i to obtain L_{i+1} ensures that the inclusion map from L_{i+1} to L_i is a valid reduction.

Theorem 5.12. *Suppose $L \not\in \texttt{P}$ is computable. Then there exists a language $K \not\in \texttt{P}$ such that $K \leq_p L$ and $L \not\leq_p K$.*

Proof. Define:

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

where f is a function we will construct later. Let $(M_i)_{i \in \mathbb{Z}^+}$ be an enumeration of polynomial time Turing Machines, which in turn enumerates the languages in $\texttt{P}$. 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 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 it. On input 1^n (with $n > 1$), M_f proceeds in two stages, each lasting exactly n steps. During the first stage, M_f computes $f(0), f(1), \ldots$, until it runs out of time. Suppose the last value M_f computed at the first stage was $f(x) = k$. At the next stage, the output of M_f will either be k or $k+1$, to be determined in the second stage.

In 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 $\texttt{P}$ as follows. The goal is to find a string $z \in \Sigma^*$ such that $z \in (L(M_i) \triangle K)$. We enumerate such strings z in lexicographic order, and then computing $M_i(z), M_L(z)$, 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 z 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 and need to do so on the next iteration).
- **Case 2:** Suppose that $k = 2i - 1$. Here, we diagonalize against polynomial-time computable functions. In this case, M_f searches for a string $z \in \Sigma^*$ such that F_i is an incorrect Karp reduction on z. That is, either:
  - $z \in L$ and $F_i(z) \not\in K$; or
  - $z \not\in L$ and $F_i(z) \in K$.

  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 $M_{\texttt{SAT}}$ is not taking too long. If such a string 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) $K \not\in \texttt{P}$, and

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

Claim 1: $K \leq_p L$.

Proof. We note that as $K \subset L$, the inclusion map $\varphi : K \to L$ sending $\varphi(x) = x$ is a polynomial time reduction. □

Claim 2: $K \not\in \texttt{P}$.

Proof. Suppose to the contrary that $K \in \texttt{P}$. Then there exists a polynomial time Turing Machine M_i that decides K. By Case 1 in the second stage of the construction of M_f, no string z was found satisfying $z \in K$ and $z \not\in L(M_i)$. 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 can be decided in polynomial time, it follows that L can be decided in polynomial time. So $L \in \texttt{P}$, a contradiction. □

Claim 3: $L \not\leq_p K$.

Proof. Suppose to the contrary that $L \leq_p K$. So there exists a polynomial-time computable function F_i from L to K. So $f(n)$ will be even for only finitely many n, which implies that K is finite. Thus, K is polynomial-time decidable, and so $K \in \texttt{P}$. This implies that $L \in \texttt{P}$, a contradiction. □

□

Remark: We note that, under the assumption that $\texttt{P} \neq \texttt{NP}$, Theorem 5.12 implies Theorem 5.14, using $L = \texttt{SAT}$. Theorem 5.12 also implies the strong version Ladner's Theorem, providing an infinite strict heirarchy of $\texttt{NP}$-Intermediate languages. The key proof technique involves applying Theorem 5.12 and induction.

Theorem 5.13 (Ladner (Strong), 1975). *Suppose $\texttt{P} \neq \texttt{NP}$. 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) $L_i \not\in \texttt{P}$ for each $i \in \mathbb{N}$.

(c) L_i is not $\texttt{NP}$-Complete for each $i \in \mathbb{N}$.

(d) $L_i \not\leq_p L_{i+1}$.

Proof. The proof is by induction on $n \in \mathbb{N}$. We let L_0 be the language constructed in Theorem 5.14. Fix $k \geq 0$ and suppose the languages $L_0, L_1, \ldots, L_k \in \texttt{NP}$-Intermediate and satisfy:

$$L_k \subsetneq L_{k-1} \subsetneq L_{k-2} \subsetneq \ldots \subsetneq L_1 \subsetneq L_0,$$

as well as that $L_i \not\leq_p L_{i+1}$ for each $0 \leq i \leq k-1$. We now apply Theorem 5.12, using L_k to obtain $L_{k+1} \subsetneq L_k$ such that $L_{k+1} \not\leq_p L_k$ and $L_{k+1} \not\in \texttt{P}$. As L_k is not NP-Complete, it follows that L_{k+1} is not NP-Complete. $\square$

5.6.1 Russell Impagliazzo's Proof of Ladner's Theorem

We conclude by providing an alternative proof of Theorem 5.14, the weak Ladner's Theorem. Ladner's original proof worked by blowing holes in SAT to construct a language that was not NP-Complete. Care was taken not to blow too many holes in SAT, resulting in the new language belonging to P. Impagliazzo's proof instead works by starting SAT instances of length n and padding these instances with strings of length $f(n) - |n|$, so that the new language L is no longer polynomial-time decidable. Note that the function $f(n)$ will be defined in the proof of Ladner's Theorem.

Observe that if $f(n)$ is a polynomial, then we can reduce SAT to L in polynomial time, simply by appending the desired suffix. This would imply that L is NP-Complete. Similarly, if $f(n)$ is exponentially large, then we have enough room to employ a brute force search to find a solution for the SAT instance φ. So L can be decided in time $\texttt{poly}(f(n))$, which places $L \in \texttt{P}$. So care needs to be taken so that $f(n)$ is larger than a polynomial, but still sub-exponential.

We now offer Impagliazzo's proof of the weak version of Ladner's Theorem.

Theorem 5.14 (Ladner (Weak), 1975). *If P $\neq$ NP, then there exists a language $L \in \texttt{NP} \setminus \texttt{P}$, such that $L \not\in$ NP-Complete.*

Proof (Impagliazzo). We define:

$$L := \{\varphi 0 1^{f(n)-n-1} : \varphi \in \texttt{SAT} \text{ and } |\varphi| = n\},$$

We note that if $f(n)$ can be computed in time poly(n), then $L \in \texttt{NP}$. Let $(M_i)_{i \in \mathbb{Z}^+}$ be an enumeration of deterministic, polynomial-time, clocked Turing Machines, where the Turing Machine M_i runs in time at most $k^i + i$, where k is the length of the input to M_i. Note that $(M_i)_{i \in \mathbb{Z}^+}$ in turn enumerates the languages of P. We define $f(n) = n^{g(n)}$, where $g(n)$ is defined as follows.

(a) $g(1) = 1$.

(b) Suppose $g(n-1) = i$. We enumerate the strings x of length at most $\log(n)$. If there exists such a string $x \in L(M_i) \triangle L$, then we set $g(n) = i + 1$. Otherwise, we set $g(n) = i$.

We note that $f(n)$ is polynomial-time computable if and only if $g(n)$ is polynomial time computable. So we prove that $g(n)$ is polynomial-time computable.

Claim 1: $g(n)$ is polynomial-time computable.

Proof. The proof is by induction on $n \in \mathbb{Z}^+$. We note that for the base case of $n = 1$, $g(1) = 1$. So $g(1)$ is polynomial-time computable in n. Now fix $k \geq 1$ and suppose that $g(k)$ is computable in time poly(k). We now show that $g(k+1)$ is polynomial-time computable. In order to compute $g(k+1)$, we first compute $g(k)$, which takes time poly(k) by the inductive hypothesis. Next, we enumerate at most all strings of length at most $\log(k+1)$. There are $2^{\mathcal{O}(\log(k+1))} = (k+1)^{\mathcal{O}(1)}$ such strings. By the construction of g, we are searching for a string $x \in L(M_i) \triangle L$. We analyze the time complexity of checking if $x \in L(M_i)$ and $x \in L$.

- We note that M_i is a polynomial time decider, which is clocked to run in time $|x|^i + i$. We note that $|x| \leq \log(k+1)$, and so M_i runs in time at most $(\log(k+1))^i + i$ on any string we are considering in the computation of $g(k+1)$.
- We now analyze the complexity of verifying that $x \in L$. Note that in a SAT instance of size $\log(k+1)$, there are at most $2^{\lceil \log(k+1) \rceil} \leq k+2$ possible instances to check. Now if x is of the form:

$$x = \varphi 0 1^{f(\log(k+1)) - |\log(k+1)| - 1},$$

then $|\varphi| < \log(k+1)$, and so we need to examine at most $k+2$ possible instances to verify whether φ is a valid instance of SAT. We note that:

$$\begin{aligned} f(\log(k+1)) - |\log(k+1)| - 1 &\leq 2f(\log(k+1)) \\ &= 2(\log(k+1))^j, \end{aligned}$$

for some $j \leq i$. So if $|x| \leq \log(k+1)$, we can check if $x \in L$ in time:

$$k + 2 + 2(\log(k+1))^j.$$

So the runtime of searching through all strings of length at most $\log(k+1)$ is bounded above by:

$$(k+1)^{\mathcal{O}(1)}\bigg((\log(k+1))^i + i + k + 2 + 2(\log(k+1))^j\bigg),$$

and the runtime of computing $g(k+1)$ is bounded above by:

$$\text{poly}(k) + (k+1)^{\mathcal{O}(1)}\bigg((\log(k+1))^i + i + k + 2 + 2(\log(k+1))^j\bigg).$$

So $g(k+1)$ can be computed in polynomial time. It follows by induction that $g(n)$ is polynomial time computable. □

As $g(n)$ is polynomial-time computable, we have that $f(n)$ is polynomial-time computable. We next show that $L \in \texttt{NP}$.

Claim 2: $L \in \texttt{NP}$.

Proof. Take $x \in L$. So x is of the form:

$$x = \varphi 0 1^{f(n)-n-1},$$

where $\varphi \in \texttt{SAT}$ and $|\varphi| = n$. Suppose we are given a satisfying instance $y_1, \ldots, y_k$ for φ as our certificate. As $\texttt{SAT} \in \texttt{NP}$, we may use the polynomial-time verifier for SAT to check that $\varphi(y_1, \ldots, y_k) = 1$. Now as $f(n)$ is polynomial-time computable, $f(n) - n - 1$ is polynomial-time computable. It remains to check that x is of the form prescribed by L; that is, x is of the form:

$$\varphi 0 1^{f(n)-n-1}.$$

This check takes polynomial-time in $f(n)$. So our check takes polynomial-time in $|x|$. It follows that $L \in \texttt{NP}$. □

Claim 3: $L \not\in \texttt{P}$.

Proof. Suppose to the contrary that $L \in \texttt{P}$. Then $L = L(M_i)$ for some i. By assumption, the runtime of M_i is bounded above by $n^i + i$. So there exists $h, k \in \mathbb{Z}^+$ such that $f(n) = n^h$ for all $n \geq k$. So there exists a polynomial-time reduction from SAT to L, mapping $\varphi \mapsto \varphi 0 1^{f(n)-n-1}$. This contradicts the assumption that $\texttt{P} \neq \texttt{NP}$. □

Claim 4: L is not NP-Complete.

Proof. Suppose to the contrary that L is NP-Complete. Then there exists a polynomial-time reduction ψ from SAT to L. We provide a polynomial-time algorithm for deciding SAT, which contradicts the assumption that $\texttt{P} \neq \texttt{NP}$. We note that as ψ is polynomial time computable, $|\psi(x)| \leq |x|^c$ for some fixed constant $c > 0$. From the proof of Claim 3, we have that $g(n)$ is unbounded. So there exists an $n_0 \in \mathbb{Z}^+$ such that $g(n) > c$ for all $n \geq n_0$. Let S be the set of strings of length less than n_0. As S is finite, we can test whether the members of S belong to SAT in constant (and therefore polynomial) time.

Now suppose φ is a string of length $n \geq n_0$. We apply $\psi(\varphi)$. If $\psi(\varphi)$ is not of the form $\tau 0 1^{f(m)-m-1}$ with $|\tau| = m$, then we have that $\varphi \not\in \texttt{SAT}$. So suppose $\psi(\varphi)$ is of the form $\tau 0 1^{f(m)-m-1}$, where again $|\tau| = m$. Note that $\varphi \in \texttt{SAT}$ if and only if $\tau \in \texttt{SAT}$. Now as:

$$\begin{aligned} |\psi(\varphi)| &= |\tau 0 1^{f(m)-m-1}| \\ &\leq |\varphi|^c \\ &= n^c. \end{aligned}$$

we have that $|\tau| < f(m) \leq |\varphi|^c = n^c$. We now argue that $m = |\tau| < |\varphi| = n$. Suppose to the contrary that $m \geq n \geq n_0$. So $g(m) > c$, which implies that $f(m) = m^{g(m)} > m^c \geq n^c$, which contradicts the fact that $f(m) \leq n^c$. It follows that $m < n$. So now we recurse on τ, applying $\psi(\tau) = \tau_1 01^{f(\ell)-\ell-1}$ and checking if $\tau_1 \in \texttt{SAT}$. The base case occurs when we arrive at an instance of $\texttt{SAT}$, of length less than n_0. Such an instance belongs to S. Recall that as S is finite, we can test whether the members of S belong to $\texttt{SAT}$ in constant (and therefore polynomial) time.

Now observe that we apply the reduction ψ at most $n - n_0 + 1$ times. Each application of the reduction takes at most n^c steps. Analyzing the base case takes time $\mathcal{O}(1)$ steps. So the algorithm has runtime at most:

$$(n - n_0 + 1)n^c + \mathcal{O}(1),$$

which is certainly polynomial in n. As we can decide $\texttt{SAT}$ in polynomial-time, it follows that $\texttt{SAT} \in \texttt{P}$, contradicting the assumption that $\texttt{P} \neq \texttt{NP}$. □

□

5.7 PSPACE

TODO

5.8 PSPACE-Complete

TODO