

Jesus Rafael Rijo

Dr. A. Digh

CSC 499.301

22 May 2021

Integrative Connections of Set Theory in Computer Science

ABSTRACT

Set theory is one of the core subfields of mathematics, providing important notions and terminology many technical fields and other mathematical subfields use. Specifically, the foundation of computer science greatly relies on the ideas and postulates established by set theory. This report outlines the essential concepts of set theory and explores some of the important intersections between set theory and computer science.

1. BASIC CONCEPTS OF SET THEORY

1.1 Key Definitions

Set theory primarily studies the properties of **sets**. A **set** can be informally defined as an unordered collection of objects called **elements/members** [41, 85]. The **cardinality** of a set describes the number of elements of a set, where $|A|$ is the cardinality of a set A . Moreover, there is a special set called the **empty set** $\emptyset = \{ \}$, where $|\emptyset| = 0$ [19, 6]. For a set A , if all the elements of A belong to another set B , then A is a **subset** of B , denoted $A \subseteq B$ [19, 12]. Based on the definition of the empty set, the empty set is a subset of all sets [19, 12]. Based on the notion of cardinality, sets can either be **finite** sets or **infinite** sets. Likewise, two sets A and B are **equal** if and only if they have the same elements, which implies that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$ [19, 5]. A given set A can also be described in terms of its **universal set** U ,

where $A \subseteq U$. For example, the set of all regular polygons is a subset of the universal set of all polygons. Moreover, sets can be expressed in several formats [35, 116]:

- **Roster method:** $W = \{1, 2, 3, 4, 5\}$; W is a set with five numbers.
- **Set-builder notation:** $\mathbb{P} = \{n \in \mathbb{N} : n \text{ is prime}\}$; \mathbb{P} is the set of all prime numbers.
- **Interval notation:** $\mathbb{R} = (-\infty, \infty)$; \mathbb{R} is the set of all real numbers.

Although sets can represent many collections of objects, including collections that lack mathematical significance, mathematicians generally focus on sets used in core subfields of mathematics, such as sets that represent **numbers systems** [19, 30]. For instance, the set of natural numbers \mathbb{N} satisfies certain properties that are essential to crucial mathematical ideas, such as summations and forms of mathematical inductions. Similarly, several technical fields, such as number theory and cryptography, thoroughly study and apply the properties of the set of integers \mathbb{Z} . [41, 76].

1.2 Set Operations

Set operations take two or more sets and produce a new set; set operations that combine two sets are **binary set operations** [41, 95-101]. Let A and B be sets in the universe U , where $A \subseteq U$ and $B \subseteq U$. Then,

- **The union of A and B :** The set $A \cup B = \{x : x \in A \text{ or } x \in B\}$.
- **The intersection of A and B :** The set $A \cap B = \{x : x \in A \text{ and } x \in B\}$.
- **The difference of A and B :** The set $A - B = \{x : x \in A \text{ and } x \notin B\}$.
- **The Cartesian product of A and B :** The set $A \times B = \{(a, b) : a \in A \text{ and } b \in B\}$; an element of $A \times B$, say $(a, b) \in A \times B$, is called an **ordered pair**.

- **The Cartesian square of A :** The Cartesian product $A \times A \times \dots \times A = \{(x_1, x_2, \dots, x_n) : x_n \in A, n \in \mathbb{N}\}$; an element of $A \times A \times \dots \times A$, say $(x_1, x_2, \dots, x_n) \in A \times A \times \dots \times A$, is called an **ordered n -tuple**.
- **The complement of A :** The set $A^C = U - A$.
- **Symmetric difference of A and B :** The set $A \Delta B = (A - B) \cup (B - A)$.

1.3 Venn Diagram

In 1881, the mathematician John Venn introduced the **Venn diagram** as a convenient method to visualize set-theoretic relationships between two or more sets [44]. A **primary Venn diagram** showcases two or more sets without asserting any claims about these sets (see Figure 1). In order to describe any set-theoretic relationships between these sets, Venn proposed shading certain areas of the Venn diagram corresponding to a given set-theoretic relationship.

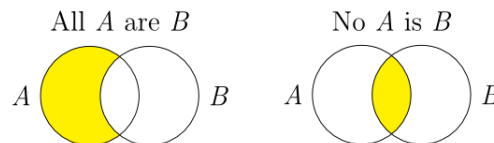


Figure 1: Two Venn diagrams representing the difference and intersection set operations, respectively [44].

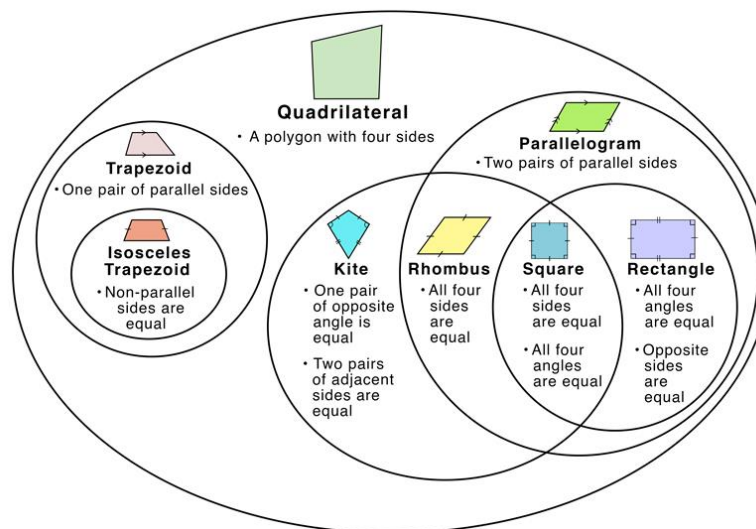


Figure 2: A Venn diagram describing different types of quadrilaterals [22].

1.4 Laws of Set Operations

Sets operations satisfy several laws that are analogous to the **logical equivalencies** found in **Boolean algebra** (see Figure 3) [35, 129-130]. These **identities** allow mathematicians to simplify complex expressions into equivalent, but simpler expressions. In order to prove an identity $A = B$, we must show $A \subseteq B$ and $B \subseteq A$ [41, 98-99].

Identity Laws	Domination Laws	Idempotent Laws	Complementation Law	Commutative Laws
$A \cap U = A$ $A \cup \emptyset = A$	$A \cup U = U$ $A \cap \emptyset = \emptyset$	$A \cup A = A$ $A \cap A = A$	$(A^c)^c = A$	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Associative laws		Distributive laws		
$A \cup (B \cap C) = (A \cup B) \cap C$ $A \cap (B \cup C) = (A \cap B) \cup C$		$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$		
De Morgan's Laws	Absorption Laws		Complement Laws	
$(A \cap B)^c = A^c \cup B^c$ $(A \cup B)^c = A^c \cap B^c$	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$		$A \cup A^c = U$ $A \cap A^c = \emptyset$	

Figure 3: Laws of set operations [35, 129-130].

1.5 Important Axioms in Set Theory

Axioms are statements that mathematicians assume to be true and can be used to prove theorems [35, 81]. The following axioms are deemed the core axioms of set theory [18]:

- **Axiom of extension:** Two sets are equal if and only if they have the same elements.
- **Axiom of specification:** For every set A and every sentence $S(x)$, there exists a set B whose elements are exactly those elements $x \in A$ for which $S(x)$ holds.
- **Axiom of pairing:** For any two sets, there exists a set that contains these two sets.

- **Axiom of unions:** For every collection of sets, there exists a set that contains all the elements that belong to at least one of the sets of said collection.
- **Axiom of powers:** For each set there exists a collection of sets that contains among its elements all the subsets of the given set.
- **Axiom of infinity:** There exists a set that contains 0 and all the natural numbers.
- **Axiom of choice:** The Cartesian product of a non-empty family of non-empty sets is non-empty.

1.6 Binary Relations

A **binary relation** R from set A to set B is a subset of $A \times B$. If $(a, b) \in R$, then a is **related** to b , denoted $a R b$ [41, 153-155]. Relations are used to describe equations, inequalities, expressions, graphs, among other mathematical ideas. Similarly, relations can be described in set-builder notation, tables, arrow diagrams, and graphs. All relations have a corresponding **domain** and **range**, where $Dom(R) = \{x \in A : \text{there is some } y \in B \text{ such that } x R y\}$ and $Rng(R) = \{y \in B : \text{there is some } x \in A \text{ such that } x R y\}$. Moreover, there are several relations that are crucial in various mathematical subfields [41, 154-158]:

- **Identity relation on A :** $I_A = \{(a, a) \in A \times A : a \in A\}$.
- **Inverse of a relation R from A to B :** $R^{-1} = \{(y, x) : (x, y) \in R\}$.
- **The composite of two relations R and S :** $S \circ R = \{(a, c) : \text{there is some } b \in B, \text{ such that } (a, b) \in R \text{ and } (b, c) \in S\}$; $S \circ R$ is not equal to $R \circ S$.

As hinted by the definition of the composite of two relations, relations can be formed among n -many sets. An **n -ary relation** R is a subset of $A_1 \times \cdots \times A_n$, for $n \in \mathbb{N}$. The sets A_1 to A_n are the **domains** of R and n the **degree**. N -ary relations are widely applied in databases

following a **relational data model**, where tables represent relations and rows/columns tuples, respectively [16].

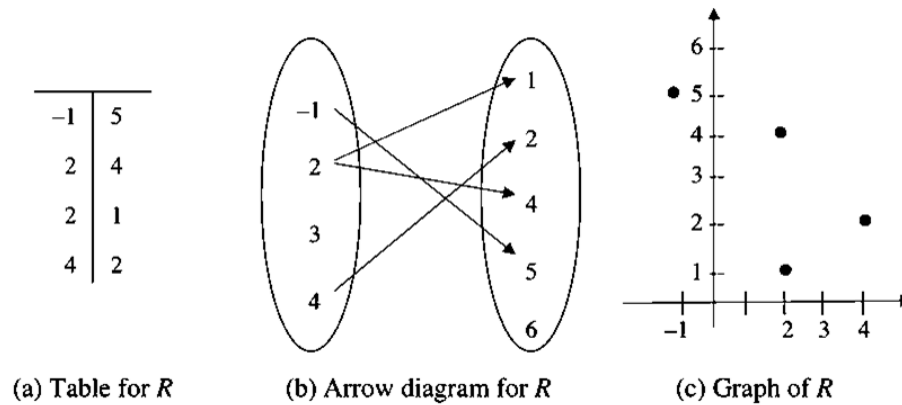


Figure 4: Different representations of the relation $R = \{(-1,5), (2,4), (2,1), (4,2)\}$ [41, 165].

1.7 Equivalence Relations and Equivalence Classes

An **equivalence relation** \sim on some set A is a special relation that satisfies these properties (see Figure 5) [41, 163-164]:

- **Reflexivity:** For all $x \in A$, $x R x$.
- **Symmetry:** For all $x, y \in A$, if $x R y$, then $y R x$.
- **Transitivity:** For all $x, y, z \in A$, if $x R y$ and $y R z$, then $x R z$.

Theorem 1: The relation $R = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} : x = y\}$ is an equivalence relation [20, 1].

Proof. Let $x \in \mathbb{Z}$. Then, $x = x$, which implies that R is reflexive. Let $x, y \in \mathbb{Z}$ and assume $x R y$. Then, since $x R y$, then $x = y$, which implies that $y = x$ and $y R x$. Thus, R is symmetric. Let $x, y, z \in \mathbb{Z}$ and assume $x R y$ and $y R z$. Then, since $x R y$ and $y R z$, $x = y$ and $y = z$, which implies $x = z$ and $x R z$. Thus, R is transitive. Therefore, R is an equivalence relation. ■

An important consequence of any equivalence relation on a set A is that said equivalence relation divides A into subsets of related elements called **equivalence classes**, where an equivalence class is denoted as $\bar{x} = \{y \in A : x R y\}$ [41, 166-167]. Moreover, the idea

of equivalence relations and equivalence classes underlies modular arithmetic [41, 182]. Thus, equivalence relations partition sets into smaller sets.

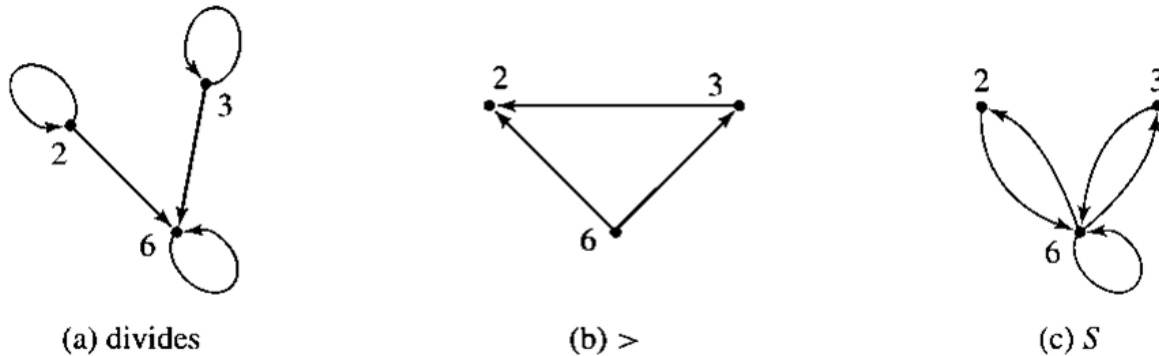


Figure 5: Three digraphs showcasing the properties of three relations [41, 165].

1.8 Functions

A **function** f from set A to set B , denoted $f: A \rightarrow B$, is a special type of relation from A to B , where B is the **codomain of f** , that satisfies these properties [41, 203-204]:

- $Dom(f) = A$
- If $(x, y), (x, z) \in f$, then $y = z$.

Furthermore, functions can be labeled based on additional conditions on the second coordinates the function [41, 222-231]:

- **Surjection:** A function $f: A \rightarrow B$, where $Rng(f) = B$; this condition says that every output of f has an input.
- **Injection:** A function $f: A \rightarrow B$, where $f(x) = f(y)$ implies that $x = y$; this condition says that every output is used at most once by an input.
- **Bijection:** A function $f: A \rightarrow B$ that is both surjective and injective.

Most branches of mathematics study and apply functions and their properties. For instance, if there exists at least one bijective function between two sets A and B of the same cardinality, then these two sets are **equivalent**, denoted $A \approx B$ [41, 262].

2. A BRIEF OVERVIEW OF THE HISTORY OF SET THEORY

2.1 Georg Cantor and Richard Dedekind: The Founders of Modern Set Theory

Although the concept of sets as collections of objects has existed alongside other primeval mathematical ideas, such as **counting** and **logic**, historians attribute the start of modern set theory to the work of the mathematicians **Georg Cantor** and **Richard Dedekind** [13]. Cantor and Dedekind ushered set theory as a mathematical subfield through their work in real analysis, which implicitly relied upon set theory. Likewise, the contributions of Cantor and Dedekind encouraged other mathematicians to define all mathematical branches in terms of set theory [13]. However, various mathematicians uncovered the flaws of early set theory by dissecting several **paradoxes** underlying the postulates of this new subfield. These discoveries led mathematicians to restrict and refine the core ideas of set theory.

2.2 Cantor's Diagonal Argument

One of Cantor's important contributions to mathematics were **countable** and **uncountable** sets. A set S is **countable** if S is finite or there exists a bijection $f: \mathbb{N} \rightarrow S$, implying that S can be listed using natural numbers. [41, 273]. If set S is infinite and there does not exist a bijection from \mathbb{N} to S , then S is **uncountable**. Cantor utilized the definition of countable sets and bijections to show that \mathbb{R} , the set of all real numbers, is an uncountable set. Cantor proved this idea through his **diagonalization argument** [49].

Theorem 2: The set of all real numbers \mathbb{R} is not countable [49].

Proof. Let $f: \mathbb{N} \rightarrow \mathbb{R}$. Then, list some of the first values of f , where each i -th row contains the decimal expansion of $f(i)$, where $i \in \mathbb{N}$:

$$f(1) = 0.13456781 \dots$$

$$f(2) = 0.53667953 \dots$$

$$f(3) = 0.95319167 \dots$$

$$f(4) = 0.34451781 \dots$$

$$f(5) = 0.23245675 \dots$$

...

Then, highlight the digits along the main diagonal:

$$f(1) = 0.\textcolor{red}{1}3456781 \dots$$

$$f(2) = 0.5\textcolor{red}{3}667953 \dots$$

$$f(3) = 0.95\textcolor{red}{3}19167 \dots$$

$$f(4) = 0.344\textcolor{red}{5}1781 \dots$$

$$f(5) = 0.2324\textcolor{red}{7}675 \dots$$

...

These digits form the decimal expansion $0.13357 \dots$. If we add 1 to each written digit of this decimal expansion, the decimal expansion becomes $0.24468 \dots$. However, list of $f(i)$ values does not contain $0.24468 \dots$ because it differs from $f(i)$ in its i -th digit. Thus, f cannot be surjective because the range of f is not equal to \mathbb{R} . Therefore, there does not exist a bijection between \mathbb{N} and \mathbb{R} . Therefore, \mathbb{R} is an uncountable set. ■

Cantor's diagonal argument has been generalized to prove important statements in other technical fields, such as formal logic and the theory of computing. For instance, Alan

Turing employed diagonalization to prove the existence of **undecidable problems**, problems that classical computers cannot process [26, 167-168].

2.3 Naïve Set Theory and Axiomatic Set Theory

Set theory can be divided into two fields: **Naïve set theory** and **axiomatic set theory**.

Naïve set theory can be defined as a lax version of set theory that describes sets using informal language and common intuitions regarding sets as any collections of elements [1, 52-53].

However, the lack of precision in naïve set theory leads to glaring contradicting outcomes regarding its key principles, such as the idea that *any* collection of elements can be a set. On the other hand, **axiomatic set theory** restricts the core principles of set theory and precisely describes them using propositional logic (see Figure 6).

Axiom of Extension: Two sets are equal if and only if they have the same elements.



$$\forall x, y: (x = y) \Leftrightarrow (\forall z: z \in x \Leftrightarrow z \in y)$$

Figure 6: The axiom of extension and its equivalent sentence in propositional logic [1, 58].

3. PARADOXES IN SET THEORY

3.1. What Is a Paradox?

A **paradox**, in the mathematical context, is “...a [statement for] which, from premises that look reasonable, one [employs] apparently acceptable reasoning to derive a conclusion that seems to be contradictory” [41, 2]. A statement like “This sentence is false” is a paradox, since said statement being true implies it is also false. Paradoxes plague most subfields of mathematics, including set theory. Specifically, **self-reference paradoxes**, statements that refer to themselves, are a common type of paradoxes in set theory [5]. Paradoxes in early set theory

were discovered when mathematicians dissected the implications of the core principles of the subfield.

3.2 Russell's Paradox

In 1901, the mathematician Bertrand Russell tackled a consequence of the notion that sets could contain any elements. Because sets could contain any elements, Russell asked if a set could contain itself. Specifically, Russell presented a set R , where $R = \{x: x \text{ is a set and } x \text{ is not an element of itself}\} = \{x: x \notin x\}$ [27, 1]. Assume that R is an element of itself. Then, by the definition of R , $R \notin R$. However, this outcome contradicts the assumption that $R \in R$. This paradox, commonly known as **Russell's Paradox**, showcases one of the flaws of early set theory [11, 3].

Russell's Paradox led many mathematicians to believe that set theory could not be used to represent the major areas of mathematics. To overcome the implications of this paradox, mathematicians decided to restrict the definition of a set by not letting any set contain itself. Therefore, Russell's Paradox exemplifies how discovering contradictions within mathematical systems helps mathematicians understand misconceptions that may arise within said mathematical systems.

4. APPLICATIONS OF \mathbb{Z} IN COMPUTER SCIENCE

4.1 Integer Representations

In order to store and manipulate information, the signals interpreted by most modern computers use just two discrete values, establishing a **binary number system** [29, 4]. A binary digit is called a **bit**; computers group bits to represent different types of information, such as symbols and program instructions. Nevertheless, computers could use a number system with

more than two values. However, a computer using more than two values would “...require complex and costly electronic circuits” and “the [computer’s] output could be disturbed by small interference voltages...” [29, 5]. Therefore, despite the simplicity of the binary number system, this cost-effective number system provides digital computers consistent reliability without limiting their performance and computational power.

Although digital computers use binary as their number system to, most modern computers support different number systems, efficiently expressing integers in different number systems (see Figure 7). For example, most computers process and manipulate **decimal** values by “...[converting] all arithmetic operations in binary and [converting] the binary results back to decimal...,” allowing users to work in the conventional decimal number system [28, 71-72]. Moreover, digital computers can directly perform arithmetic operations with decimal numbers by converting the decimal numbers into **binary codes**. A **binary code** is a group of n bits that supports 2^n unique combinations of 1’s and 0’s, such that each combination represents one element of the given decimal number. Digital computers compute binary code by using different **bit assignment** methods, which determine the size and number of bit groups. Many digital computers incorporate the **binary-coded decimal (BCD)** method in order to generate binary codes. This bit assignment method converts decimal numbers to binary codes by assigning four bits to each decimal digits (see Figure 8). Likewise, digital computers handle data with letters and various special characters. Digital computers follow **alphanumeric binary codes**, such as the American Standard for Information Interchange (*ASCII*), to interpret data containing numbers, letters, and special characters [28, 73].

```

procedure base_b_expansion( $n, b \in \mathbb{N} : b > 1$ )
     $q := n$ 
     $k := 0$ 
    while  $q \neq 0$ 
         $ak := q \bmod b$ 
         $q := q \operatorname{div} b$ 
         $k := k + 1$ 
    return ( $ak-1, \dots, a1, a0$ ) { ( $ak-1 \dots a1a0$ )b }

```

Figure 7: An algorithm to compute the base b expansion of an integer n [35, 249].

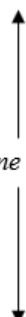
<u>Decimal number</u>	<u>Binary-Coded Decimal Number</u>	
0	0000	 Code for one decimal digit
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

Figure 8: A table listing some decimal numbers and the respective binary-coded decimals [28, 73].

4.2 Big Integer Class

Most programming languages offer **primitive data types** that dictate the values variables can store and the operations that can be performed on these variables. One common characteristic among all primitive data types is the maximum and minimum values that restrict the size of a variable. For instance, the "... `int` data type [in Java] is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31} - 1$ "

[31]. However, certain computations involve integers that surpass the maximum and/or minimum values imposed by a programming language. Java provides a **BigInteger class** as an approach to store and manipulate large integers. Java's BigInteger class overcomes the limits of the primitive data types by storing an integer as an array of type `int` [10]. Likewise, the BigInteger Class not only has the arithmetic operations offered by the `int` data type, but also provides efficient functions "...for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation..." [10]. However, the standard libraries of other programming languages, such as C++, lack an equivalent class to Java's BigInteger class. Nonetheless, the basic implementation of a BigInteger class only requires a data structure that stores the number's digits, sign, and high-order digit (see Figure 9).

```
#define MAXDIGITS 100 //Maximum length of array
#define PLUS 1 //Positive/negative sign bits
#define MINUS -1
struct bignum {
    char digits[MAXDIGITS];
    int signbit; //PLUS or MINUS
    int lastdigit; //High-order index
};
```

Figure 9: A C++ struct that represents a BigInteger [42, 104].

4.3 Mathematical Induction Over Positive Integers

Various mathematical statements hold for all positive integers, where \mathbb{Z}^+ represent the set of positive integers. This property is described as **the principle of mathematical induction**. The principle of mathematical induction considers some set S that describes a statement, where $S \subseteq \mathbb{N} = \mathbb{Z}^+$. Then, $S = \mathbb{N}$ if S satisfies these properties [41, 115].

- $1 \in S$.
- For all $n \in \mathbb{N}$, if $n \in S$, then $n + 1 \in S$.

The principle of mathematical induction is used in **inductive definitions**, where we describe a first object and describe the $(n + 1)$ st object in terms of the n th object [41, 116].

For example, the sum of n terms $\sum_{i=1}^n x_i$ can be inductively defined as:

- $\sum_{i=1}^1 x_i = x_1$.
- For all $n \in \mathbb{N}$, $\sum_{i=1}^{n+1} x_i = \sum_{i=1}^n x_i + x_{(n+1)}$.

Moreover, the principle of mathematical induction underlies many proofs of **program correctness**, where a **correct** program produces the correct output for every possible input [35, 372].

4.4 Recursion and Recursive Definitions

Recursion describes the process of defining an object in terms of itself. Mathematics and computer science use recursion primarily through **recursive definitions**, which can describe functions, sequences, sets, among other mathematical structures (see Figure 10). We can describe a recursively-defined function $f(x)$, with domain \mathbb{N} , by using a **basis step** and **recursive step** to define its range [35, 345]:

- **Basis step:** Specifies the value of the function at some integer x .
- **Recursive step:** Outlines a rule to find $f(n)$ based on f evaluated at smaller integer values.

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ n \cdot F(n - 1), & \text{if } n > 1 \end{cases}$$

Figure 10: A recursively-defined function $F: \mathbb{N} \rightarrow \mathbb{N}$ that finds the factorial of a natural number n [Amir, 2].

Likewise, a **recursive algorithm** “...solves a problem by reducing it to an instance of the same problem with smaller input,” integrating recursively-defined functions in the algorithm design process [35, 361]. Recursive algorithms are primarily used with data structures that we generally define recursively. For example, a **tree** can be recursively defined “...as a collection of nodes...with a root and zero or more nonempty **subtrees**, each of whose roots are connected by a directed edge from [the root]” [48, 121]. Due to the recursive nature of trees, many operations involving trees, such as traversals, insertions, and deletions, are also defined recursively.

However, certain correct recursive algorithms may exhaust the memory provided by the **function call stack**, a dynamic entity that maintains the local variables and parameters during the execution of a function [24]. **Stack overflow** describes the instance in which a function goes beyond the memory constraints during runtime, leading the program to end abruptly. Programmers can use **recursion trees** to visualize how the number of recursive calls increases as the input increases (see Figure 11). Therefore, some recursive algorithms

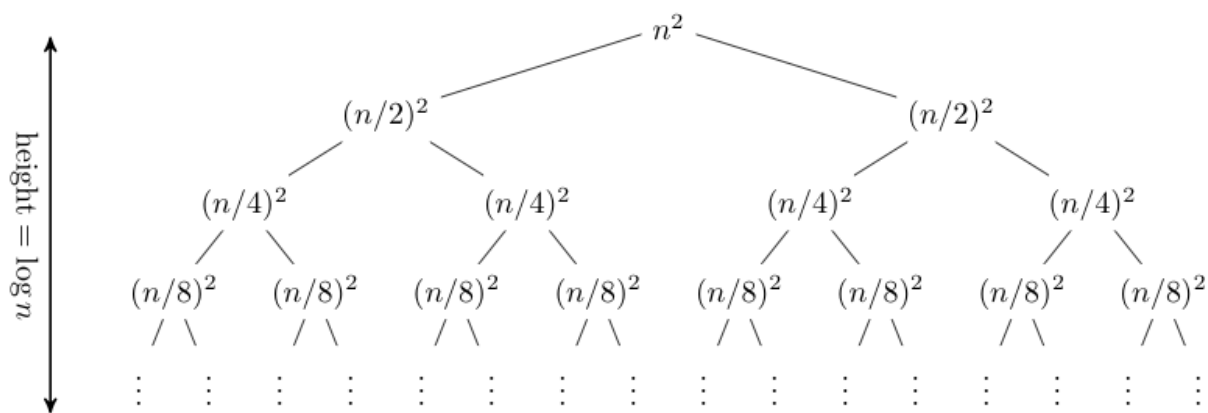


Figure 11: A recursion tree for the recursively-defined function $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ [14].

4.5 Bit Sets

A **bit set** is a data structure that stores bits, emulating an array of Boolean elements. Bit sets are optimized for improved space allocation. In a bit set, “...each element [generally] occupies only one bit...,” meaning that a bit set’s element occupies eight times less memory than the smallest primitive type `char` [3]. Furthermore, most bit set implementations allow the user to access and modify each bit position. However, unlike other sequential data structures, a bit set cannot be dynamically resized during runtime because its size is fixed at compile time. Bit sets are often used to manage groups of Boolean flags, where variables can represent any combination of flags (see Figure 12) [23, 650]. Likewise, several libraries with bit set embellish the utility of bit sets. For example, the C++ `<bitset>` library provides “...the ability to convert integral values into a sequence of bits, and vice versa, [through a temporary bit set]” [23, 652].

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

Figure 12: A bit set that represents the computer science courses taken by a student [47, 52].

4.6 Bit Masking

Programmers can manipulate the individual bits of a bit set with **bitwise operators**. **Bitwise operators** consider each bit of the bit set, treating them as binary variables [28, 108]. The following bitwise operators are commonly use to manipulate individual bits or groups of bits [4]:

- **Bitwise-AND:** Performs a logic AND operation on a bit-by-bit basis.
- **Bitwise-OR:** Performs a logic OR operation on a bit-by-bit basis.

- **Bitwise-XOR:** Performs an exclusive OR operation on a bit-by-bit basis.
- **Left shift:** Shifts the bits of the first operand to the left.
- **Right shift:** Shift the bits of the first operand to the right.
- **Complement:** Complements bits on a bit-by-bit basis.

A **bit mask** is a constructed bit sequence we can use with bitwise operators to manipulate specific bits in a bit set. For example, programs can use bitwise-OR and bitwise-AND operators alongside specific bit masks to switch certain bits on and off respectively (see Figure 13).

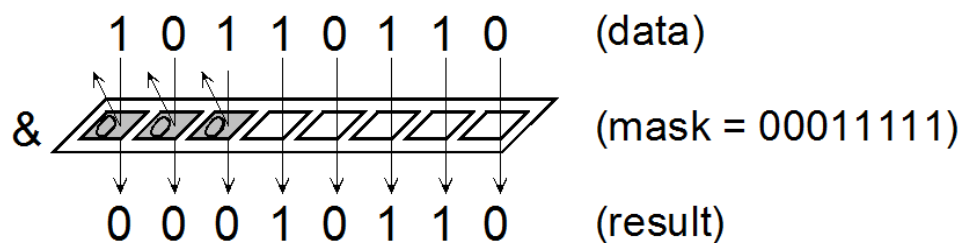


Figure 13: A bitwise-AND between $(10110110)_2$ and the bit mask $(00011111)_2$ turns off the three leftmost bits of the original bit set [4].

4.7 Enumerated Data Types

Enumeration is the process of listing all the members of a set by using another set, such as the set of natural numbers \mathbb{N} or a subset of \mathbb{N} . **Enumeration** is one of the main subfields of combinatorics, encompassing key concepts like **generating functions** and **recurrence relations** [25, 1-3]. Programming languages apply enumeration through **enumerated datatypes**.

Enumerated datatypes in C allows programmers to create custom data types, where each custom type receives a name and a list of labels initialized to unique integer values that cannot mutate during running time [McPherson, 1]. Enumerated datatypes often improve programs by

“... [making them] more readable..., providing a way to incorporate meaningful labels...,”

simplifying future debugging, and facilitate self-documentation (see Figure 14) [McPherson, 2].

```
enum ParityType { EVEN, ODD };

ParityType FindParity( int integer )
{
    if( integer % 2 == 0 ) {
        return EVEN;
    }
    else {
        return ODD;
    }
}
```

Figure 14: A C enumerated datatype (`enum`) used by a function to return the parity of an integer [McPherson, 2].

5. THE C++ SET DATA STRUCTURE

5.1 Can a Computer Represent All Sets?

Computers can represent many sets and perform set operations on these sets.

Nonetheless, computers cannot completely represent all sets, like the set of all real numbers \mathbb{R} .

Nonetheless, computers can solve certain problems regarding some of the sets they cannot fully represent, such as **membership problems**, problems in which a computer verifies that a particular element belongs to a given set. For example, an algorithm cannot store the set \mathbb{P} of primes to show a given number is a prime because \mathbb{P} is infinite by Euclid’s Theorem [8]. Instead, computers solve membership problems involving \mathbb{P} through **primality tests**, algorithms that apply a deterministic or probabilistic approach to show a given number belongs to \mathbb{P} [7].

5.2 The C++ Set Data Structure

The C++ set data structure is a STL container that orders elements based on a sorting criterion. The implementation of the C++ set allows the user to use the default sorting criterion

or define a sorting criterion. Moreover, all sorting criteria must outline a **strict weak** ordering, such as $<$ (“strictly less than”), a relation that satisfies these properties [43]:

- **Irreflexivity:** For all $x \in A$, $x \not R x$.
- **Antisymmetry:** For all $x, y \in A$, if $x R y$, then $y \not R x$.
- **Transitivity:** For all $x, y, z \in A$, if $x R y$ and $y R z$, then $x R z$.

Based on these properties, a relation such as \leq (“less than or equal to”) does not satisfy the sorting criterion for sets. Likewise, the sorting criterion for C++ sets “...is used to check equivalence, where elements are considered to be duplicates if neither is less than the other” [23, 315]. Sets are node-based containers that can be implemented as balanced binary trees [23, 325]. Nonetheless, certain implementations choose red-black trees over balanced binary trees because red-black trees guarantee, for any tree of size n , a maximum height of $2 \log_2(n + 1)$ and logarithmic search time [48, 566-567].

5.3 Basic Set Modifiers

The C++ set provides several operations to modify the elements contained in an set object [39]:

- **insert:** Adds one or more unique elements to a set object.
- **erase:** Removes an element or a range of elements between the set’s first element and last element.
- **swap:** Swaps the elements of two sets of the same type; this function works for sets with different cardinalities.
- **clear:** Removes all the set’s elements, reducing the set’s cardinality to zero.

5.4 The Union, Intersection, Difference, and Symmetric Union of Two Sets

The union, intersection, difference, and symmetric union operations are offered by the **merge operations** in the `<algorithm>` standard library. These operations accomplish the same results as the analogous set operations. However, these merge operations only work for sorted ranges. Thus, the contents of all set objects must be sorted in order to apply any of the four merge operations.

5.5 The Cartesian Product of Two Sets

The C++ set does not provide a Cartesian product operation. Implementing a Cartesian product operation of two finite sets A and B would involve an algorithm that generates all the ordered pairs of the set $A \times B$. Since $|A \times B| = |A||B|$, a basic implementation of the Cartesian product would have a running time of $O(|A||B|)$ (see Figure 15).

```
vector< vector<string> > cartesianProduct( vector<string> set_A,
vector<string> set_B ) {
    vector< vector<string> > cartesianProduct( set_B.size() );
    string entry;
    for( int i = 0; i < set_A.size(); i++ ) {
        for( int j = 0; j < set_B.size(); j++ ) {
            entry = "(" + set_A[i] + ", " + set_B[j] + ")";
            cartesianProduct[i].push_back( entry );
        }
    }
    return cartesianProduct;
}
```

Figure 15: A C++ function that computes the Cartesian product of two sets represented as vectors.

6. COMPUTING THE POWER SET OF A FINITE SET

6.1 The Power Set of a Set

The power set of a set A , denoted $\wp(A)$, is the set whose elements are the subsets of A and $|\wp(A)| = 2^{|A|}$ [41, 90-91]. The cardinality of the power set can be proved using induction

or through the **product rule**, one of the core techniques of **counting**. The product rule states that “...if there are n consecutive tasks to do, the tasks are independent, and there are a_i ways to do the i th task, then there are exactly $a_1 a_2 \dots a_n$ ways to complete the sequence of n tasks” [41, 140].

Theorem 3: If a set A is a set with n elements, then $\wp(A)$ is a set with 2^n elements.

Proof. Let A be a set, where $|A| = 0$. Thus, $A = \emptyset$. Then, $\wp(A) = \{\emptyset\}$ and $|\wp(A)| = 1 = 2^0$. Thus, the theorem holds for $n = 0$. Let $A = \{a_1, a_2, \dots, a_n\}$ be a set, where $|A| = n$ and $n \in \mathbb{N}$. To describe some $B \subseteq A$, we must determine whether each $x_i \in B$, where $i \in \mathbb{N}$. For each x_i , either $x_i \in B$ or $x_i \notin B$. Then, by the Product Rule, there are $2 \cdot 2 \cdot 2 \cdot 2 \cdot \dots \cdot 2 = 2^n$ of building any arbitrary subset of A . Therefore, $|\wp(A)| = 2^{|A|}$. ■

One glaring difficulty of efficiently generating $\wp(A)$ is that $|\wp(A)|$ grows exponentially as A grows, implying that computing the powerset of large finite sets can become unfeasible in terms of running time and space complexity (see Figure 16).

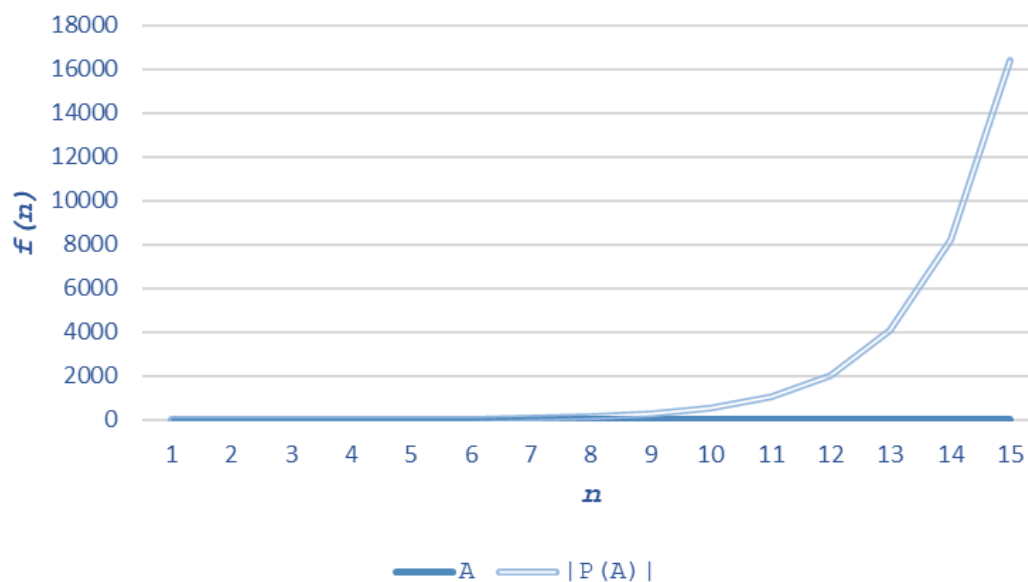


Figure 16: A graph illustrating how $|\wp(A)|$ grows exponentially as $|A|$ grows linearly.

6.2 Computing $\wp(A)$ with a Decision Tree

A **decision tree** is a method of **supervised learning** in artificial intelligence in which “at each node of a tree, a test is applied which sends the query sample down one of the branches of the node” [45, 2-4]. This process continues until the algorithm reaches a leaf. We may apply the concept of a decision tree to the generation of the powerset of a set A . First, the algorithm starts with its index pointing to the first element of A . Then, the algorithm makes two recursive calls, where both calls move the index to the next element of A . One of the recursive calls adds the first element into the subset it produces, while the other call does not add said element. The algorithm continues this process recursively until the indices reach the end of A and store the subsets of A into $\wp(A)$ (see Figure 17). For a set A , this algorithm will make $2^{|A|}$ recursive calls to generate $\wp(A)$.

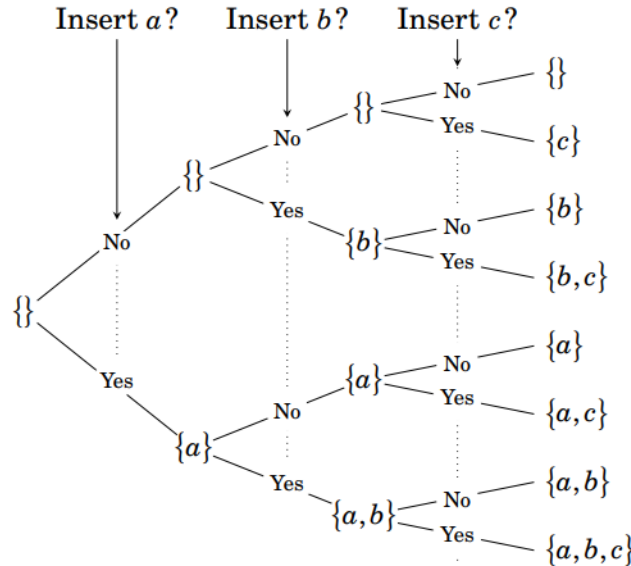


Figure 17: A decision tree for $\wp(A)$, where $A = \{a, b, c\}$ [19, 13].

6.3 Computing $\wp(A)$ with Backtracking

Backtracking is an algorithm design technique that attempts to cleverly apply an exhaustive search, where the algorithms “...recursively [evaluate] every alternative and then chooses the best [decision]” [12, 71]. For instance, programs that play strategic games, such as checkers and chess, heavily use backtracking algorithms [48, 511-512]. The driver program of a backtracking algorithm represents the solution as a vector $a = (a_1, \dots, a_k)$, where each element in a comes from a finite ordered set S [42, 167]. At each step, the backtracking algorithm starts with partial solution with k -many elements, with $k \leq n$, such that the driver program never tries one possibility more than once. The backtracking algorithm attempts to reach the final solution by adding another element to the vector of the partial solution. If the algorithm confirms the current vector is the final solution, then the program stops seeking solutions. Else, the algorithm determines whether the current partial solution can be extended to a complete solution or the added element must be removed to find this solution (see Figure 18).


```

bool finished = FALSE;
backtrack(int a[], int k, data input)
{
    int c[MAXCANDIDATES];
    int ncandidates;
    int i;
    if (is_a_solution(a,k,input))
        process_solution(a,k,input);
    else {
        k = k+1;
        construct_candidates(a,k,input,c,&ncandidates);
        for (i=0; i<ncandidates; i++) {
            a[k] = c[i];
            backtrack(a,k,input);
            if (finished) return;
        }
    }
}

```

Figure 18: Driver program for the backtracking algorithm [42, 167].

Additionally, backtracking algorithms typically utilize three subroutines [42, 168-169]:

- **is_a_solution(a, k, input)**: This subroutine tests whether the first k elements of a vector a are a complete solution for the given problem. `input` stores general information for the subroutine, such as the size of a target solution.
- **construct_candidates(a, k, input, c, ncandidates)**: This subroutine populates a vector c with the complete set of possible candidates for the k th position of a . `ncandidates` denotes the number of candidates returned, whereas `input` specifies auxiliary information.

- `process_solution(a, k)`: This subroutine processes a complete solution.

```

is_a_solution(int a[], int k, int n)
{
    return (k == n);
}

construct_candidates(int a[], int k, int n, int c[], int
*ncandidates)
{
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}

process_solution(int a[], int k)
{
    int i;
    printf("{");
    for (i=1; i<=k; i++)
        if (a[i] == TRUE) printf(" %d",i);
    printf(" }\n");
}

```

Figure 19: Subroutines to compute with $|p(a)|$ backtracking in C [42, 169-170].

7. COMPUTING THE CARDINALITY OF A FINITE POWER SET

7.1 A Note About Exponentiation

In certain instances, we may want to find the cardinality of the power set of a finite set A , denoted $|p(a)| = 2^{|A|}$. The process of computing $|p(a)|$ is known as **exponentiation**, the procedure of finding or approximating the value of the expression b^n , where b^n means the repeated multiplication of b $(n - 1)$ -many times, for $n \in \mathbb{N}$. Likewise, the study and

implementation of efficient **exponentiation** algorithms are crucial in several technical fields, particularly cryptography.

7.2 Naïve Exponentiation

The task of computing $|\wp(A)| = 2^{|A|}$ can be solved using **naïve exponentiation**. Naïve exponentiation is an exponentiation method that computes b^n , where b is the **base** and n the **exponent**, by multiplying b by itself $n - 1$ times (see Figure 19) [30]. However, since n determines the number of multiplications, naïve exponentiation becomes inefficient for large values of n .

```
int exp(const int n, const int b)
{
    int num = 1;
    for(int i = 0; i < n; i++) {
        num *= b;
    }
    return num;
}
```

Figure 19: A C++ function to compute b^n with naïve exponentiation.

7.3 Repeated Squaring

Repeated squaring is an exponentiation method that speeds up the process of finding $|\wp(A)|$ by reducing the number of multiplications. This algorithm first squares base b to obtain b^2 . Then, the algorithm takes b^2 to obtain $(b^2)^2 = b^4$ and iteratively continues until reaching the target power of form $b^{(2^n)}$, where n denotes the number of square operations (see Figure 20) [30]. Repeated squaring has a running time of $O(\log_2(n))$ because the algorithm requires $\log_2(n)$ multiplications to exponentiate. However, repeated squaring only computes powers

in the form $b^{(2^n)}$ (see Figure 20). Despite the improved running time over naïve exponentiation, this particular implementation of the repeated squaring method can be deemed unpractical.

```
long fast_exponentiation(const long n, const long b)
{
    long num;
    if( n == 0)
    {
        return b;
    }
    else
    {
        num = b * b;
        for(int i = 0; i < n; i++)
        {
            num *= num;
        }
        return num;
    }
}
```

Figure 20: A C++ function to compute b^n with repeated squaring.

7.5 Exponentiation with Arithmetic Shift-Left

An **arithmetic shift** on a binary number is an operation that shifts a binary number to the left or right. An **arithmetic shift-left** multiplies a binary number by 2, while an **arithmetic shift-right** divides the number by 2 [28, 114]. Since $|\wp(A)| = 2^{|A|}$ can be represented as a binary number, then $|\wp(A)|$ can be computed by taking binary number $(1)_2$ and applying **arithmetic shift-left** operations $|A|$ -many times. However, arithmetic shift operations may yield mathematically incorrect results in certain scenarios. For example, if

arithmetic operations exceed the number of bits of a primitive `int` data type, then the result will be arithmetically incorrect [37]. Nevertheless, programmers can overcome the constraints of primitive data types by utilizing data structures with optimized memory allocation, such as bit sets and vectors of Boolean values.

7.6 Exponentiation with Precomputation and the BGMW Method

Certain exponentiation techniques speed up the computation process by precomputing some of the powers [15, 140]. The **Brickell, Gordon, McCurley, and Wilson (BGMW) method** first explored the idea of incorporating precomputed values in exponentiation. The simplest iteration of the BGMW method first stores the powers g^{2^k} , $\exists x \in \mathbb{N}$, and then computes the final power with **binary method**, which computes b^n by using the binary expansion of n [15, 132-133]. Nonetheless, precomputing powers requires extra memory space, which may discourage precomputation when storage constraint are present.

7.7 Approximating $|\wp(A)|$ with Taylor's Polynomials

Programmers can approximate computationally-demanding functions, like $|\wp(A)| = 2^{|A|}$, by approximating these functions with polynomials, one of the simplest types of functions to compute in terms of running time and space complexity. We can represent a function $f(x)$ by using its **Taylor series** at some value a , such that [46, 759-761]:

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(a)}{i!} (x - a)^i$$

Because computers cannot compute the exact value of an infinite series, we can approximate $f(x)$ by computing the **n th-degree Taylor polynomial** of $f(x)$ at a , the n th partial sum of the Taylor series [46, 774]:

$$f(x) = \sum_{i=0}^n \frac{f^{(i)}(a)}{i!} (x-a)^i$$

The difference between the approximation and the exact value of $|\wp(A)|$ decreases as n increases (see Figure 21). Likewise, these approximations will have a respective **round-off error**, the error that occurs “...because the arithmetic performed [by a program] involves...approximate representations of the actual numbers” [6, 17-18].

	$x = 0.2$	$x = 3.0$
$T_2(x)$	1.220000	8.500000
$T_4(x)$	1.221400	16.375000
$T_6(x)$	1.221403	19.412500
$T_8(x)$	1.221403	20.009152
$T_{10}(x)$	1.221403	20.079665
e^x	1.221403	20.085537

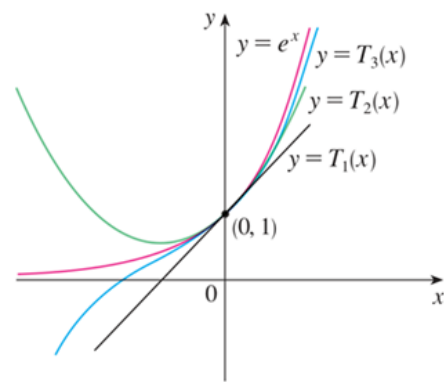


Figure 21: A table comparing the outputs of different n th polynomials of $f(x) = e^x$ and their graphs.

8. DISJOINT SETS: AN APPLICATION OF SET THEORY

8.1 Dynamic Equivalence Problems

Given a finite set F and two elements $x_1, x_2 \in F$, the premise of a **dynamic equivalence problem** is to determine if x_1 and x_2 are equivalent, where equivalence describes the properties of an equivalence relation [48, 352]. Moreover, the equivalences among elements dynamically change depending on the user's input. For example, **dynamic graph connectivity** is a dynamic equivalence problem where the connectivity between nodes can change arbitrarily [38, 5-6]. In this kind of dynamic equivalence problem, the program must be capable of forming

connections among nodes, destroying connections among nodes, and determining if two or more nodes are connected.

8.2 Overview of Disjoint Sets

Disjoint sets are a data structure that represents dynamic equivalence problems [48, 353-355]. Disjoint sets use an array of integers to represent n -many disjoint sets, where all entries of the array are initialized to -1 to denote that each entry represents an equivalence class. Thus, every disjoint sets object starts with n -many equivalence classes. A disjoint sets object starts as a forest of n -many trees of size 1. After all disjoint sets are merged as one set, the forest becomes a single tree of size n (see Figure 22). In order to represent disjoint sets, a disjoint sets class must offer these basic operations [48, 355-357]:

- **union:** Makes two elements equivalent by modifying the array.
- **find:** Searches the array to determine if an equivalence class contains a particular an element.
- **Deunion:** Undoes the equivalence between two sets by modifying the array.

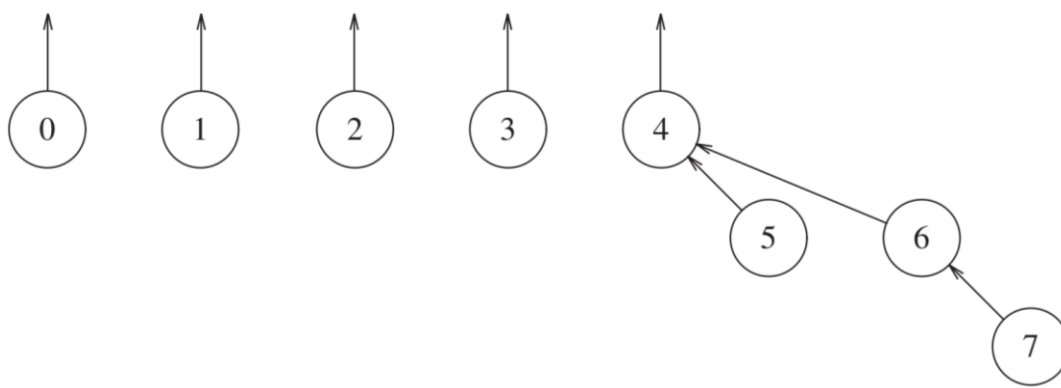


Figure 22: A disjoint sets object of size 8 after several arbitrary union operations [48, 358].

8.3 *find* Operation

A `find` operation takes the index of a particular element to determine which set contains the element, where the root of the set denotes the name of the set. If the element at the index is equal to -1 , then the function returns the index itself, which says that the element belongs to its original set. On the other hand, if the element at the index is not equal to -1 , then the function calls itself recursively, using the element at the given index as its new index (see Figure 23). The running time to compute a `find` operation depends on the depth of the tree containing the element. Thus, the running time of a `find` operation is for a disjoint sets object of size n is $O(n)$.

```
int find( int x ) const
{
    if( s[ x ] < 0 )
    {
        return x;
    }
    else
    {
        return find( s[ x ] );
    }
}
```

Figure 23: Basic implementation of a `find` operation [48, 358].

8.4 *union* Operation

A `union` operation merges two sets together into one set by turning the root of one set into the root of the other set [48, 354]. In order to verify if the sets are already united, the union operation calls the `find` operation with each set. If the sets do not share the same root,

then we can unite the two given sets. Given the running time of the `find` operation, the `union` operation has a running time of $O(n^2)$ (see Figure 24).

```
void union( int root1, int root2 )
{
    if( find( root1 ) != find( root2 ) )
    {
        s[ root1 ] = root2;
    }
}
```

Figure 24: Basic implementation of a `union` operation [48, 358].

8.5 Union-by-Size and Union-by-Rank

We can improve the depth of the trees created by several `union` calls by implementing a **union-by-size** approach, which considers the sizes of two sets before merging them together. Instead of storing the size of each set in another array, we “...can have the array entry of each root contain the negative [value] of the size of [each tree]” [48, 357-359]. Moreover, union-by-size guarantees that the depth any node will be strictly less than $\log_2(n)$ [48, 359].

8.6 Union-by-Height

Union-by-height is another approach analogous to union-by-size that also improves the depth of the trees and guarantees depths of strictly less than $\log_2(n)$ for all nodes. This algorithm considers the height of each tree before merging them together. In order to store the height of each set, the algorithm stores the $-h - 1$ on each array cell, where $-h - 1$ is the negative of the set’s height minus one [48, 358]. Therefore, union-by-height provides the same improvements as the union-by-size, meaning union-by-height is a trivial modification to union-by-size (see Figure 25).

```

void union_by_height( int root1, int root2 )
{
    if( s[ root2 ] < s[ root1 ] ) {
        s[ root1 ] = root2;
    }
    else {
        if( s[ root1 ] == s[ root2 ] ) {
            --s[ root1 ];
        }
        s[ root2 ] = root1
    }
}

```

Figure 25: Basic implementation of a `union` operation by height [48, 358].

8.7 Path Compression

The aforementioned improvements to the performance of the disjoint sets data structure have focused on optimizing the `union` operation. We could further improve disjoint sets by embellishing the `find` operation with **path compression**. If we perform a `find` with path compression on a node x , then every node on the path from x to the root has its parent recursively changed to the root (see Figure 26) [48, 360]. Although `find` with path compression can be considered a trivial change to the basic `find` operation, incorporating path compression in the case in which `union` operations are performed arbitrarily yield a maximum running time $O(M \log_2(N))$ [48, 361].

```

int find_with_compression( int x )
{
    if( s[ x ] < 0 )
    {
        return x;
    }
    else
    {
        return s[ x ] = find_with_compression( int x );
    }
}

```

Figure 26: Basic implementation of a `find` operation with path compression [48, 358].

9. GENERATING MAZES WITH DISJOINT SETS

9.1 Basic Concepts of Graph Theory

Most maze generation methods are established on the basic concepts of **graph theory**.

Graph theory is a subfield of mathematics that gained prominence during the first half of the 1900s [9, xi]. Graph theory focuses on the properties and applications of **graphs**. A **graph** $G = (V(G), E(G))$ is made of two sets $V(G)$ and $E(G)$, where $V(G)$ is the **vertex set** and $E(G)$ the **edge set** [9, 2]. Each edge of a graph is a pair (u, v) , where $u, v \in V(G)$ [48, 379-380]. Mazes can be represented using **simple undirected graphs**. A **simple undirected graph** is a graph “...in which each [undirected] edge connects two different vertices and no two edges connect the same pair of vertices” [35, 642-643]. Since the graph of a maze is analogous to a matrix of length l and width w , the number of vertices $|V(G)| = lw$ (see Figure 27).

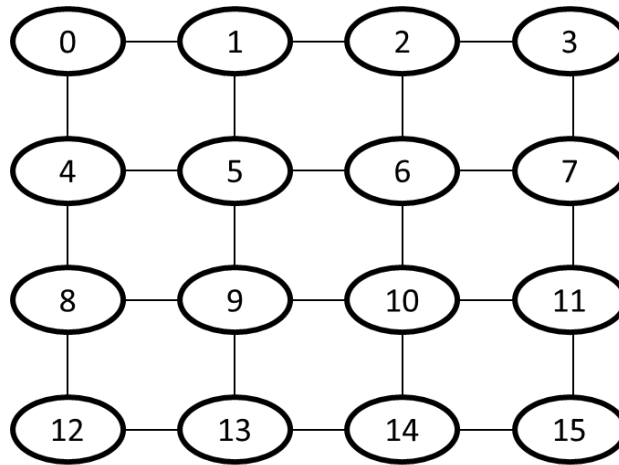


Figure 27: A representation of a 4×4 grid as an undirected graph with 16 vertices [19, 13].

9.2 Maze Generator: An Overview

A maze generator first takes the length l and width w of the grid in order to create an undirected graph G with lw vertices and zero edges. Then, the maze generator initializes a disjoint sets object with lw sets. To populate the edge set, the maze generator will randomly choose two vertices and create an edge between them, as long as the edge satisfies the geometry of the lw matrix. For each `union` operation, the maze generator will check if all sets have been united. If the maze generator detects that all sets are united, then the edge set is complete, such that all vertices are reachable. Because there are two `find` calls per each randomly chosen edge, the driver program will call roughly $2(lw)$ to $4(lw)$ `find` calls. Therefore, the running time of the maze generator is approximately $O(n \log_2(n))$, where $n = lw$ [48, 373].

9.3 Pseudo-Random Numbers

The maze generator should ideally perform unions based on **randomly-selected** sets, which increases the variety among the mazes. However, computers cannot truly generate random numbers. Steve Ward, Professor of Computer Science and Engineering at

Massachusetts Institute of Technology, explains that computers follow intricate algorithms that attempt to imitate true randomness [36]. Therefore, random number generators produce **pseudorandom numbers**, numbers that appear to be random and follow specific statistical properties [48, 494-495]. For example, the **linear congruential generator** is a simple random number generator that produces a sequence of integers that satisfy $x_{(i+1)} = Ax_i \bmod(M)$. In order to start the sequence, the user first choose a **seed**, the initial value x_0 [48, 496]. Many programming languages offer random number generators in their standard libraries. In C++, the `<random>` and `<cstdlib>` libraries provide customizable random number generation [32].

10. SET THEORY IN AUTOMATA THEORY

10.1 What Is the Theory of Computing?

The **theory of computing** is one of the foundational subfields of computer science. The theory of computing studies **computation**, the alteration and/or movement of data [26, xiii]. By analyzing the properties of computation, the theory of computing attempts to answer central questions regarding the capabilities, efficiency, and limitations of classical computers. The theory of computing is divided into three areas [40, 1-3]:

- **Automata Theory:** Automata theory outlines the formal definitions and properties of **automata**, the mathematical models of computation.
- **Computability Theory:** Computability theory classifies problems into two categories, studying their intrinsic properties: **Solvable** problems and **unsolvable** problems.
- **Complexity Theory:** Complexity theory studies computable problems based on their difficulties, analyzing the inherent properties that make certain problems more difficult than other problems.

The subfields of the theory of computing represent the foundation of modern computer systems and provide crucial concepts applied in several non-theoretical subfields of computer science.

10.2 Basic Definitions of Automata Theory

Automata theory focuses on the study of automata, which are mathematical representations of computers. The ideas studied in automata theory generally involve **strings**, **alphabets**, and **languages** [26, 7]:

- **String:** A finite sequence of symbols where each symbol is chosen from an alphabet; every string w has a specific length $|w|$.
- **Alphabet:** A finite set of symbols, such as the binary alphabet $\{0, 1\}$.
- **Language:** A finite or infinite set of strings.

Moreover, automata are categorized based on their **computational power**, which determines what tasks a certain type of automata can do. A **finite automaton** can only compute a few primitive functions and lacks the computational power to perform intricate tasks, such as string manipulation and sorting. On the other hand, a **Turing machine** satisfies the capabilities that any computer algorithm could require, implying that every computer algorithm can be implemented as a Turing machine [26, 121-122].

10.3 Deterministic Finite Automata: An Example of Automata

Deterministic finite automata (DFA) are one of the simplest models of computation studied in automata theory. A DFA is a device with no auxiliary memory that has several states and a processing unit with limited memory [26, 13]. A DFA changes state by taking input from a tape one character at time, where reading a character could change the current state of the

finite state machine. Finite automata can be represented with finite state diagrams, a digraph “...where [the] nodes represent states and arrows are labeled with characters from the input alphabet” (see Figure 28) [26, 15-16]. Although DFA are a versatile computational model, DFA lack the computational power to modify inputs. Thus, DFA can determine if an input string belongs to a language. Nonetheless, DFA can be used to solve other computational problems beyond pattern recognition, despite their limited computational power of finite automata. For instance, the behavior of a vending can be represented through an intricate DFA that accounts the different purchases a buyer can make (see Figure 29).

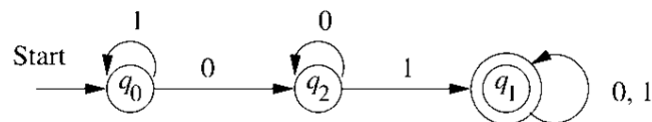


Figure 28: A state transition diagram for a DFA that accepts strings with the substring 01 [21, 48].

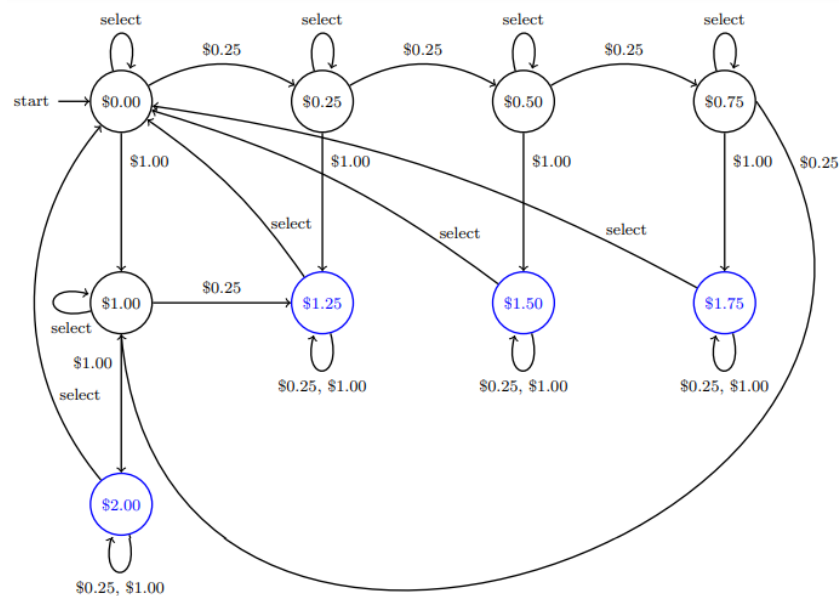


Figure 29: A DFA that represents the basic operations of a vending machine [17, 4].

In order to accurately describe the characteristics and computational power of any automaton, said automaton must be formally described with set theory [26, 14]. Therefore, a DFA is mathematically denoted as a quintuple $A = \{Q, \Sigma, \delta, s, F\}$, where:

- Q : A finite set of states.
- Σ : A finite input alphabet.
- δ : A transition function, defined as $\delta: Q \times \Sigma \rightarrow Q$.
- s : The initial state of the automaton; $s \in Q$.
- F : A finite set of favorable states $F \in Q$.

Based on this definition, a **configuration** (s_M, c) of a DFA M is an element of $K \times \Sigma^*$ that describes the current state of the DFA s_M and the remaining input c of an input string w [34, 41-42]. A **computation** by M is a finite sequence of configurations $C_0, C_1, \dots, C_n, n \in \mathbb{N}$. The **computation** for a string w determines if M accepts or rejects w :

- If the computation ends in a **favorable configuration**, then M accepts w .
- If the computation ends in a **rejecting configuration**, then M rejects w .

Theorem 4: Every DFA M , when taking input w , will halt after $|w|$ steps [34, 44].

Proof. Recall that M performs n computations to process w , where C_n denotes the final computation. C_n is either an accepting or a rejecting configuration, implying that M will halt after reaching C_n . Since each computation consumer one character of w , $n = |w|$. Therefore, M will halt after $|w|$ steps. ■

10.4 Regular Expressions

Despite being a concept within the theory of computing, **regular expressions** are widely embedded in several subfields of applied computer science. Formally, a **regular expression** is a

string that can be formed based on these rules, where α and β are arbitrary regular expressions [34, 92]:

- \emptyset is a regular expression.
- ϵ is a regular expression.
- $\forall w \in \Sigma$ are regular expressions.
- $\alpha \cdot \beta$ is a regular expression.
- $\alpha \cup \beta$ is a regular expression.
- α^* is a regular expression.
- α^+ is a regular expression.
- (α) is a regular expression.

With regular expressions, we can represent **regular languages**, languages that can also be represented with finite automata. Therefore, every regular expression has a corresponding regular language, and vice versa, as described by **Kleene's theorem** [34, 97-98]. With regular expressions, programmers can define regular expressions that represent string patterns. For example, the regular expression (a^*b^*) corresponds to the regular language $L = \{a^n b^m : n, m \in \mathbb{N}\}$ [26, 34-35]. On the other hand, the regular expression $([0-9]\{1,3\}(\backslash. [0-9]\{1,3\})\{3\})$ recognizes valid IP addresses [34, 107]. Many programming languages provide libraries that define and apply regular expressions. `<regex>`, the C++ standard library for regular expressions, provides tools for pattern search and string replacement with regular expressions [33].

Theorem 5: The set of languages that can be defined with regular expressions is exactly the set of regular languages. [33, 103].

Proof. Note that “every regular language can be defined with a regular expression” [34, 100]. Proving this statement involves constructing an arbitrary DFA M and a regular expression α , such that $L(M) = L(\alpha)$. Note that “any language that can be defined with a regular expression can be accepted by some finite state machine, [implying the language] is regular [34, 96]. Proving this statement involves constructing an arbitrary finite state machine F , such that $L(F) = L(\alpha)$. Therefore, these two statements imply that the set of all languages that can be defined with regular expressions is equal to the set of regular languages. ■

11. CONCLUSION

As an extensive mathematical subfield, many researchers continue studying set theory, tackling open problems in the area. Given the pervasiveness of set theory within computer science, any major discoveries in set theory could greatly contribute to ongoing research in theoretic computer science. Even if progress in set theory research stagnates, set theory will continue being an integral part in the understanding of advanced mathematics and computer science

Bibliography

- [1] Aspnes, James (2020). "Notes on Discrete Mathematics." *Yale University*. Retrieved from <http://www.cs.yale.edu/homes/aspnes/classes/202/notes.pdf>.
- [2] Bagaria, Joan (2020). "Set Theory." *The Stanford Encyclopedia of Philosophy (Spring 2020 Edition)*. Retrieved from <https://plato.stanford.edu/entries/set-theory/>.
- [3] "Bitset." *cplusplus.com*. Retrieved from <http://www.cplusplus.com/reference/bitset/bitset/>.
- [4] Brinkerhoff, Delroy A. (2021). "2.11. Supplemental: Bitwise Operators." *Object-Oriented Programming Using C++*. Retrieved from <https://icarus.cs.weber.edu/~dab/cs1410/textbook/index.html>.
- [5] Bolander, Thomas (2017). "Self-Reference." *The Stanford Encyclopedia of Philosophy (Winter 2017 Edition)*. Retrieved from <https://plato.stanford.edu/entries/self-reference/#SetThePar>.
- [6] Burden, Richard L. and J. Douglas Faires (2010). *Numerical Analysis*, 9th ed.
- [7] Caldwell, Chris (2021). "Finding Primes & Proving Primality: How to Prove Primality (Single Page Version)." *Prime Pages*. Retrieved from <https://primes.utm.edu/prove/merged.html>.
- [8] Caldwell, Chris (2021). "Euclid's Proof of the Infinitude of Primes (c. 300 BC)." *Prime Pages*. Retrieved from <https://primes.utm.edu/notes/proofs/infinite/euclids.html>.
- [9] Chartrand, Gary, et. al (2012). *A First Course in Graph Theory*, Dover ed.
- [10] "Class BigInteger." *Java SE Documentation. Oracle*. Retrieved from <https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html>.

- [11] Dillig, Işıl. "Sets, Russell's Paradox, and Halting Problem." *CS311H: Discrete Mathematics*. Retrieved from <https://www.cs.utexas.edu/~isil/cs311h/lecture-sets-6up.pdf>.
- [12] Erickson, Jeff (June 2019). "Backtracking." *Algorithms*. Retrieved from <https://jeffe.cs.illinois.edu/teaching/algorithms/book/02-backtracking.pdf>.
- [13] Ferreirós, José (2020). "The Early Development of Set Theory." *The Stanford Encyclopedia of Philosophy (Summer 2020 Edition)*. Retrieved from <https://plato.stanford.edu/entries/settheory-early/>.
- [14] Foster, Nate (2012). "Lecture 20: Recursion Trees and the Master Method." *Cornell University*. Retrieved from <https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/lec20.html>.
- [15] Gordon, Daniel M (April 1998). "A Survey of Fast Exponentiation Methods." *Journal of Algorithms*, vol. 27(1), pp. 129-146. DOI: <https://doi.org/10.1006/jagm.1997.0913>.
- [16] Grayson, Tom (11 March 2010). "Referential Integrity and Relational Database Design." *Massachusetts Institute of Technology*. Retrieved from http://web.mit.edu/11.521/www/lectures/lecture10/lec_data_design.html.
- [17] Gribkof, Eric (2013). "Applications of Deterministic Finite Automata." *UC Davis*. Retrieved from <https://www.cs.ucdavis.edu/~rogaway/classes/120/spring13/eric-dfa.pdf>.
- [18] Halmos, Paul R (1960). *Naïve Set Theory*, The University Series in Undergraduate Mathematics.

- [19] Hammack, Richard (2018). *Book of Proof*, 3rd ed. Retrieved from <https://www.people.vcu.edu/~rhammack/BookOfProof/Main.pdf>.
- [20] Heap, Aaron. "Equivalence Relation." *The State University of New York College at Geneseo*. Retrieved from https://www.geneseo.edu/~heap/courses/330/equivalence_relation.pdf.
- [21] Hopcroft, Motwani, and Jeffrey D. Ullman (2006). *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.
- [22] Kofman, Ilya (2021). "Quadrilateral Venn Diagram." *The City University of New York*. Retrieved from <https://www.math.csi.cuny.edu/~ikofman/math329.html>.
- [23] Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*, 2nd ed.
- [24] Kauffman, Christopher (17 June 2015). "Function Call Stack Examples." *George Mason University*. Retrieved from <https://cs.gmu.edu/~kauffman/cs222/stack-demo.html>.
- [25] Keller, Mitchel T. and William T. Trotter (2015). *Applied Combinatorics*, preliminary ed. Retrieved from <https://people.math.gatech.edu/~trotter/book.pdf>.
- [26] Kinber, Efin and Carl Smith (2001). *Theory of Computing: A Gentle Introduction*.
- [27] Mann, Kathryn. "Russell's Paradox." Retrieved from <https://math.berkeley.edu/~kpmann/Russell.pdf>.
- [28] Mano, Morris (1992). *Computer System Architecture (International Edition)*, 3rd ed.
- [29] Mano, Kime, and Tom Martin (2015). *Logic and Computer Design Fundamentals*, 5th ed.
- [30] Pauli, Sebastian (2016). "Powers and Logarithms." *MAT 112 Ancient and Contemporary Mathematics, UNC Greensboro*. Retrieved from <https://mathstats.uncg.edu/sites/pauli/112/HTML/chapterpowersandlogs.html>.

- [31] "Primitive Data Types." *The Java™ Tutorials*. Oracle. Retrieved from <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.
- [32] "Random". *cplusplus.com*. Retrieved from <https://www.cplusplus.com/reference/random/>.
- [33] "<regex>". *cplusplus.com*. Retrieved from <https://www.cplusplus.com/reference/regex/>.
- [34] Rich, Elaine (July 2019). *Automata, Computability and Complexity: Theory and Applications*.
- [35] Rosen, Kenneth H. (2007). *Discrete Mathematics and Its Applications*, 7th ed.
- [36] Rubin, Jason M. (1 November 2011). "Can a computer generate a truly random number?" *MIT School of Engineering*. Retrieved from <https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/>.
- [37] "Safe Numerics." *Boost: C++ Libraries*, Silicon Graphics Computer Systems, Inc. Retrieved from https://www.boost.org/doc/libs/1_76_0/libs/safe_numerics/doc/html/introduction.html.
- [38] Schwarz, Keith (2016). "Dynamic Connectivity." *Stanford University*. Retrieved from <http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/17/Small17.pdf>.
- [39] "Set." *cplusplus.com*. Retrieved from <https://www.cplusplus.com/reference/set/set/>.
- [40] Sipser, Michael (2013). *Introduction to the Theory of Computation*, 3rd ed.
- [41] Smith, Eggen, et al. (2015). *A Transition to Advanced Mathematics*, 8th ed.

- [42] Skiena, Steven S. and Miguel A. Revilla (2002). *Programming Challenges: The Programming Contest Training Manual*.
- [43] "Strict Weak Ordering." *Boost: C++ Libraries*, Silicon Graphics Computer Systems, Inc.
Retrieved from <https://www.boost.org/cgi/stl/StrictWeakOrdering.html>.
- [44] Shin, Lemon, et. al. "Diagrams." *The Stanford Encyclopedia of Philosophy (Winter 2018 Edition)*. Retrieved from
<https://plato.stanford.edu/archives/win2018/entries/diagrams/>.
- [45] Suter, David et. al (July 2018). "Artificial Intelligence Decision Trees." *The University of Adelaide*. Retrieved from
https://cs.adelaide.edu.au/~dsuter/Harbin_course/DecisionTrees.pdf.
- [46] Stewart, James (2016). *Calculus: Early Transcendentals*, 8th ed.
- [47] Troccoli, Nick (2019). "CS107, Lecture 3: Bits and Bytes; Bitwise Operators." *Stanford Computer Science*. Retrieved from
<https://web.stanford.edu/class/archive/cs/cs107/cs107.1196/lectures/3/Lecture3.pdf>.
- [48] Weiss, Mark Allen (2014). *Data Structures and Algorithm Analysis in C++*, 4th ed.
- [49] Weisstein, Eric W. "Cantor Diagonal Method." *MathWorld--A Wolfram Web Resource*.
Retrieved from <https://mathworld.wolfram.com/CantorDiagonalMethod.html>.