SWE MASTER LIST

A Guide to Preparing for the Software Engineering Interview Process

TOMMY MONSON

CONTENTS

Preface 6
Introduction: Why? 7

◇ Theory of Computation 10

1 Automata Theory 10
1.1 Exploring the Link Between Concrete and Abstract Machines 10
1.2 Classes of Automata and the Languages They Can Understand 12
1.2.1 Finite-state Machines 14
1.2.2 Pushdown Automata 16
1.2.3 Linear Bounded Automata 17
1.2.4 Turing Machines 17
1.3 The Importance of Turing Machines to Modern Computing 20

2 Computability Theory 24
2.1 The Scope of Mathematical Problem Solving 24
2.2 Decision Problems and Function Problems 25
2.3 "Effective Calculability" 26
2.4 The Ballad of Georg Cantor 27
2.4.1 The First Article on Set Theory 27
2.4.2 Ordinals and Cardinals 29
2.4.3 The Continuum Hypothesis 29
2.4.4 Cantor's Later Years and Legacy 32
2.5 The Diagonal Argument for Computable Functions 33
2.5.1 Uncountably Many Languages 34
2.5.2 Countably Many Turing Machines 35
2.5.3 Computable Functions and Computable Numbers 37
2.6 Hilbert's Program and Gödel's Incompleteness Theorems 38
2.7 The Entscheidungsproblem and the Church-Turing Thesis 39
2.7.1 μ-recursive Functions 39
2.7.2 The Halting Problem 39
2.7.3 The Untyped λ-calculus 39
2.8 Computability is Recursion 39
2.9 Turing Degrees 39

3 Computational Complexity Theory 39
3.1 Big O Notation 39
3.2 Complexity Classes 39

◇ Programming Language Theory 40

4 Elements of Programming Languages 40
4.1 Syntax 40
4.2 Type Systems 40
4.3 Control Structures 40
4.4 Libraries 40
4.5 Exceptions 40

5 A History of Programming Languages 40

6 Programming Paradigms 40
6.1 Imperative . . . 40
6.2 Functional . . . 40

7 Higher-Order Programming 40
7.1 Formal Systems of Logic . . . 40

◇ Abstract Data Types and Data Structures 41

8 A Motivation for Objects 41

9 Abstract Data Types and the Data Structures that Implement Them 43
9.1 Lists . . . 45
9.1.1 Arrays . . . 45
9.1.2 Linked Lists . . . 46
9.1.3 Skip Lists . . . 47
9.2 Stacks . . . 47
9.3 Queues . . . 48
9.4 Deques . . . 49
9.5 Priority Queues . . . 49
9.6 Graphs . . . 49
9.7 Trees . . . 50
9.7.1 Binary Search Trees . . . 51
9.7.2 Binary Heaps . . . 56
9.7.3 Tries . . . 57
9.8 Maps . . . 58
9.8.1 Hash Maps . . . 58
9.8.2 Tree Maps . . . 59
9.9 Sets . . . 60
9.10 Multisets (or Bags) . . . 60
9.11 Monoids . . . 60

◇ Algorithms 61

10 Searching 61
10.1 Depth-First Search . . . 61
10.2 Breadth-First Search . . . 64
10.3 Bidirectional Search . . . 66
10.4 Dijkstra's Algorithm . . . 67
10.5 Binary Search . . . 67
10.6 Rabin-Karp Algorithm . . . 68

11 Sorting 68
11.1 Selection Sort . . . 68
11.2 Insertion Sort . . . 68
11.3 Merge Sort . . . 68
11.4 Quick Sort . . . 68
11.5 Radix Sort . . . 68
11.6 Topological Sort . . . 68

12 Miscellaneous 68
12.1 Cache Replacement Algorithms . . . 68

12.2 Permutations . . . 68
12.3 Combinations . . . 69
12.4 Bit Manipulation . . . 69

◇ Implementation, Featuring an Exploration of Java 70

13 How does Java work? 70

14 Java Standard Library 70
14.1 java.lang . . . 70
14.2 java.util . . . 70
14.3 java.io . . . 70
14.4 java.net . . . 70

15 Java Techniques 70
15.1 Static Initialization Blocks . . . 70
15.2 Lambda Expressions . . . 70

16 Java Frameworks 70

◇ Additional Technologies 71

17 Unit Testing 71
17.1 JUnit . . . 71

18 Version Control 71
18.1 Git . . . 71

19 Build Automation 71
19.1 Make . . . 71
19.2 Maven . . . 71

20 Unix 72
20.1 History of Unix and GNU/Linux . . . 72
20.2 A Tour of the Unix File System . . . 72
20.3 Common Commands and Tasks . . . 72
20.4 Making a Personalized Linux Installation . . . 72

21 Virtualization 73

22 Containerization 73
22.1 Docker . . . 73

23 Markup and Style Languages 73
23.1 TeX . . . 73
23.2 HMTL . . . 73
23.3 CSS . . . 73

24 Data Formats 73
24.1 XML . . . 73
24.2 JSON . . . 73

25 Query Languages 73
25.1 SQL 73

◇ Software Development 74

26 Software Engineering Processes and Principles 74

27 Design Patterns 74

◇ Appendices 75

A System Design 75
A.1 Server Technologies 75
A.2 Persistent Storage Technologies 75
A.3 Network Techniques 77

Preface

I wrote my first novel because I wanted to read it.

—Toni Morrison

Words, words, words...

Introduction: Why?

Why data structures? Why algorithms? Why computers, in general? This is a guide for learning data structures and algorithms in intense detail. So you should first ask yourself why. Why am I reading this? In one sense, for the purpose of interviewing well at top software engineering companies. Or startups. Or whatever kind of computer-type job you're looking for that involves writing code. But in another sense, why even study all of this stuff? Why do people study data structures? Why do people write algorithms? Why do we use computers?

Because humans like fast. Thousands of years ago, we liked fast access to food, so we developed agriculture. A little over a hundred years ago, we liked fast personal transportation, so we invented automobiles. Nowadays, we like fast consumption of media, so we created a digital infrastructure that allows us to consume whatever we want *on demand*. We like having computers that do cool stuff and enhance our lives. And to make computers do that, we have to tell them to move a bunch of stuff around and to do it fast. The "stuff" is a bunch of values and data structures full of values that represent *information*. The "moving around fast" is done by algorithms. The data structures are the nouns and the algorithms are the verbs. Let's learn those nouns and verbs so we can use them to instruct our computer to do cool stuff. But before we do that, let's introduce a few major ideas that will be important throughout the length of this document: data types, space complexity, and time complexity.

NOUNS

"What kind of stuff are we talking here? Data?
Numbers? Gold doubloons?"

At the physical level, we're talking about bits. Billions of electrical switches in your computer's memory, each of which can be turned on or off, 1 or 0. However, we can choose to interpret those bits in different ways and depending on how we interpret them, the same sequence of 0s and 1s could represent many different kinds of data. We are able to assign a chunk of bits a *data type*. Having types allows us to simulate all kinds of stuff like numbers or pictures or songs or whatever you can think of. Even gold doubloons.

How much stuff are we talking here? A handful?
60 boxes worth? 20 kilobytes?"

It depends on how many bits you need to represent what you're interested in. Typically, in real-world programming, we don't use a lot of singular values. Often, we use many values or *elements* contained in a *collection* of some kind, such as a list of integers. These collections typically hold many elements of the same size and type (e.g. 32 bits for a 32-bit integer). We could, then, measure the size of a collection in terms of elements instead of bits. If we have a data structure that needs to keep track of n values, how much space does it take up in terms of elements?

The answer seems obvious. If a data structure needs to keep track of n values, it will store n elements that represent those values. We would say then that the data structure has an $O(n)$ *space complexity* or that it has an algorithm which belongs to the order of algorithms known as $\mathcal{O}(n)$, where n is the number of input values. However, there actually are some data structures that require more than n elements to represent

n values. This is because they use a space-inefficient algorithm in at least one of their operations. For example, consider a skip list, which is a data structure that is similar to a linked list. It stores elements with duplicate values in order to support a fast searching algorithm. For every n values it needs to keep track of, a skip list actually stores $n \cdot \log n$ elements. It has a space complexity of $O(n \log n)$.

VERBS

"How fast are we talking here? Immediately?
A box per minute? An hour to finish?"

I'd like to put "fast" into perspective with an example. Imagine that you are a school administrator, and you need to look up student transcripts from time to time. Each transcript is labeled with a student's ID number. You have a collection of student transcripts and you need to find the one you are looking for, given the student's ID number.

This is an example of a *search problem*, one of the most commonly encountered problems in computer science. You have a bunch of things and you want a particular one of those things. How do you find it? More importantly, how do you find it quickly? This problem is solved by a particular kind of algorithm known as a *search algorithm*, of which there are many.

A search algorithm is a method of finding some value, given some key. For example, find a student's transcript, given their ID. There may be many methods that one could use to eventually find that transcript, but some methods will find it much more quickly. Naturally, the type of collection the transcript resides in is a factor in determining which methods are fast and which ones are slow to find it. This collection type also introduces an upper bound for how fast you could possibly find that transcript. Imagine the student transcripts are in a stack on a desk. You would probably have to just look through them one by one until you find what you are looking for. Imagine instead if each transcript is stored in its own drawer, and each drawer is labeled with the transcript's student ID. You would be able to find your desired transcript on the first try by opening the drawer with the ID on it. Both of these scenarios involve search algorithms, but one algorithm is much faster than the other because it was designed for a collection type that supports faster searching (drawers instead of a stack). This is where the idea of *time complexity* arises.

Time complexity, as it pertains to search algorithms, asks the following question: "Given a data structure full of items, what is the fastest reliable way that you could find a particular item in it?". The word *reliable* is important here. Let's say your transcripts are in a stack and your plan is to look through all of them. You could pick up the transcript on the top of the pile and end up finding what you are looking for. However, you cannot reliably do this. Your transcript could just as easily have been in the middle of the pile or even at the bottom, and it would have taken a lot longer to find it in these cases. Thus, it is essential to define the "fastness" of the algorithm in a way that is *state-independent*. That is, the particular order of the elements in the collection should not influence the classification of "how fast" the algorithm is. In practice, the classification is usually made according to the algorithm's *worst-case* time complexity, its efficiency when the order of the elements is the least conducive to finding the target (e.g. when the transcript is at the very bottom of the stack). In this case, we keep the state of the collection constant and discuss the time complexity in terms of a different variable: the number of input values.

Obviously, if your stack contained a million transcripts, it would take longer to search it than if it had 20 transcripts. In the worst case, you would have to look through a million transcripts instead of 20. Let's consider an arbitrary number of transcripts, n. We would say that your search algorithm, when applied to a stack of n values, has an $O(n)$ time complexity. We would also say that it is a *linear* search algorithm that belongs to the order of algorithms known as $\Theta(n)$. Other search algorithms have different time complexities and may require certain kinds of data structures or a particular ordering of elements.

In general, an algorithm could be designed to accomplish almost any task. Regardless of the nature of the task, the time complexity of an algorithm should always be evaluated in terms of the size of the input.

GOING FORWARD IN THIS GUIDE...

It's time to get specific. This guide will be more succinct and technical than this introduction, but it will remain conversational in tone. Jargon is useful, but it is often used imprecisely or as a cop-out when someone doesn't actually know what he or she is talking about. For that reason, even simple terms are defined before they are used. This review is fairly detailed and attempts to be as precise as possible without succumbing to pedantry or bloat. It is written in such a way that any curious person should be able to understand it with some supplementary Googling of unfamiliar terms.

I recommend reading this guide slowly, acknowledging each word and pausing to think about each topic deeply. This is not a bullet-pointed list of properties and brief descriptions to be memorized. The goal of this guide is instead to give you a *holistic* understanding data structures and algorithms. When the time comes to answer a whiteboard question, you will have a much higher chance of solving it if you understand data structures and algorithms *fundamentally*. If you can visualize these abstract objects in your mind, you will be able to see them for what they are: tools in your arsenal. The meat of this guide is in sections 3 and 4, the Data Structures and Algorithms sections. You can skip to them, if you like, but the first two sections are designed to give you this deep understanding that I believe is important.

[Section descriptions]

Theory of Computation

The *theory of computation* is a field of study that is related to both mathematics and computer science. It is concerned with the formalization of "models of computation" or "abstract machines", which are theoretical models that describe how the output of a mathematical function is computed given inputs. It is also concerned with the problems that can be solved by these machines and how efficiently they can be solved. The theory of computation gives us the formal definition of a computer and the scope of what a computer can accomplish.

The field has three branches: *automata theory*, *computability theory*, and *computational complexity theory*, each of which asks fundamental questions pertaining to how mathematical calculations can be automated, which kinds of problems can be solved by a machine, and how many resources are required to solve a particular problem. We will touch on the major discoveries made in each of these branches and discuss why they are significant to computer science and programming in general.

1 AUTOMATA THEORY

An *automaton* is a self-operating machine designed to follow or respond to a predetermined sequence of operations. Automata include physical devices such as cuckoo clocks or robots or computer processors as well as *abstract machines*. An abstract machine is like a theoretical computer. It is a mathematical description of what a computer is. It models the computation that a real, concrete computer (or automaton) could actually perform. Like their concrete counterparts, abstract machines have a starting *state*. The machine receives an *input* and will *transition* to another state based on the value of that input and its current state.

Abstract automata can fully model the computation that is performed by today's state-of-the-art computers, and we will spend a lot of time in this guide discussing the structure and behavior of this model as well as its limitations. We will also briefly explore the abstract automata that exist in pure mathematics and model computation that is not physically realizable.

1.1 Exploring the Link Between Concrete and Abstract Machines

Let's take a simple, concrete machine and try to visualize the abstract machine that underlies it. Consider a cuckoo clock. Every day, at noon, a small plank with a mechanical bird perched on its lip will extend from a hole above the clock face, powered by a working motor. Once the plank is fully extended, the bird will flap its metal wings and maybe turn its head as a song is played from a revolving music box inside the clock. Then, when it is 12:01 PM, the music will stop, the bird will still, and the plank will retract. The process will occur again the next day, at noon.

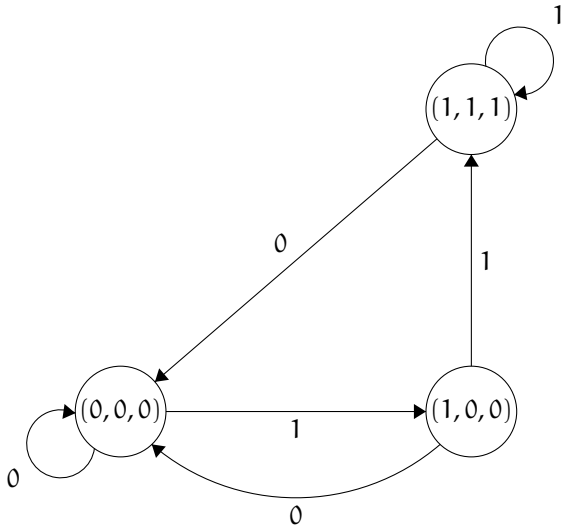
The machine's components here are the plank, the bird, and the music box. The components each have two states: the plank can be retracted or extended, the bird can be still or flapping its wings, and the music box can be paused or playing. The input to this machine is whether or not the clock hands both point straight at 12. We can say that this automaton has a state that can be expressed in terms of the states of its

components. We can represent its starting state with the sequence (0,0,0) (i.e. the plank is retracted, the bird is still, and the music box is paused). The automaton can receive one of two inputs, 1 or 0 (i.e. both clock hands point straight at 12 or they do not). This input might be implemented with an electromechanical switch that turns the motor on or off.

(0,0,0) -> (1,0,0) We start at state (0,0,0). As the morning passes, the automaton receives a constant input of 0, which keeps it in its starting state. At noon, the input switches to 1, and the automaton transitions to the state (1,0,0). As the plank extends and transitions its own state from 0 to 1, the bird remains still, and the music box remains paused. Once the plank is fully extended, we are in state (1,0,0).

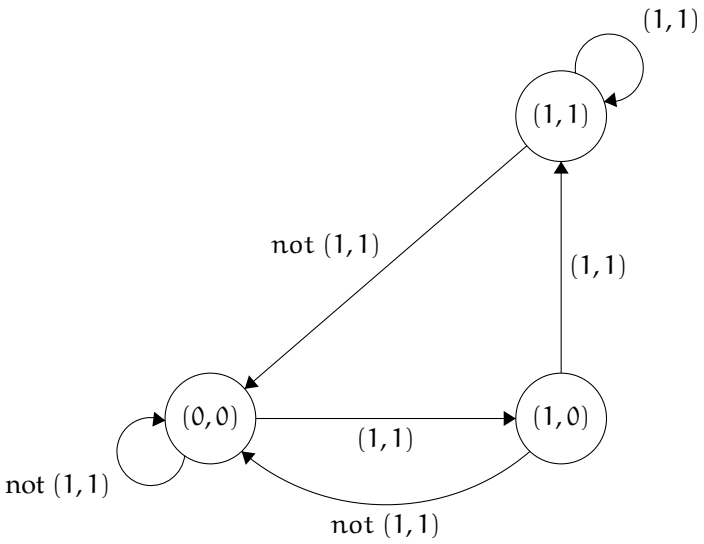
(1,0,0) -> (1,1,1) We expect that the plank will be fully extended well before 12:01 PM, so the input should still be 1. However, if for some reason the input is 0, we should transition back to (0,0,0), as the machine should only operate outside of its starting state between 12:00 and 12:01 PM. This is a simple example of *strange input*. The input 0 at this state is unexpected, but it is technically possible. It is a better idea to define the behavior for this unexpected case, rather than to allow the input to cause unexpected behavior. On the other hand, if the input is 1 at this state, the automaton transitions to the state (1,1,1). The plank is fully extended, the bird is flapping its wings, and the music box is playing.

(1,1,1) -> (0,0,0) Before 12:01 PM, the input is still 1, and that input should keep the automaton in its current state of (1,1,1). When 12:01 PM rolls around, the input will now be 0. In this case, we would like to return to the original state. On receipt of the input 0 while in state (1,1,1), the automaton transitions back to its starting state, (0,0,0). This abstract machine is shown below with a state format of (plank, bird, music) and an input specifying if the clock hands point exactly toward 12.



This is a perfectly reasonable abstract machine, but something isn't right. It does not accurately simulate the behavior of the cuckoo clock we described. What happens when the clock strikes midnight? The clock hands will point at 12, producing an input of 1, but we don't want the bird to wake everyone up at midnight. We could use two bits of input: whether or not the hands are between 12:00 and 12:01, and whether it is AM or PM. Additionally, the music never plays unless the bird is flapping its wings, so we can combine those states. The abstract machine is better represented by the diagram

below with a state format of (plank, bird/music) and an input format of (hands are toward 12, is PM).



This model of computation is called a *finite-state machine* (FSM), and we will discuss it in detail later. For now, note that the actual complexity of the concrete cuckoo clock would be a bit more complicated than the FSM above. For example, what does the bird do specifically when it is on? Perhaps it raises and lowers its wings three times, turns its head to the right twice, and repeats. This process can also be modeled by an FSM. Thus, we can have FSMs whose states are themselves FSMs. Stated another way, an FSM can model systems of any arbitrary, but finite, complexity.

1.2 Classes of Automata and the Languages They Can Understand

Automata, in general, are machines that receive, interpret, and follow instructions that are written according to the *grammar* of a *formal language* (a language whose rules can be formally defined). How powerful or expressive a language is determines what you can say using it. If we are using this language to instruct a computer to solve a problem, the *expressiveness* of the language will determine which instructions we are able to give. If we cannot express a given instruction, we will not be able to solve any problems requiring that instruction.

A Brief Introduction to Formal Grammars

A formal grammar is a finite set of *production rules* that specifies what a particular language is supposed to look like. It formalizes the structure of "grammatical units" in a language. This is done in order to enforce a specific alphabet, a consistent lexicon, and a predictable sentence structure, all of which contribute to how useful and distinct any given language is.

A grammar G consists of the following components:

- A finite set N of *nonterminal* symbols.
- A finite set Σ of *terminal* symbols that is disjoint from N.

- A symbol $S \in N$, designated as the *start symbol*.
- A finite set P of *production rules* where each rule is of the form $(\Sigma \cup N)^*N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$.

Put another way, a production rule converts a sequence of symbols to another sequence of symbols. If a rule is of the form *left-hand side* $\rightarrow$ *right-hand side*, each side can contain terminals and nonterminals, but its left-hand side must contain at least one nonterminal. A nonterminal is a symbol that is replaced by other symbols when an appropriate rule is applied. Unlike a terminal, it does not belong to the alphabet of the language, but rather represents a sequence of symbols that do. Terminals cannot be replaced by other symbols once chosen.

Let's construct a simple formal grammar. It is typical for the nonterminals to be uppercase and for the terminals to be lowercase. Let N = {SENTENCE, NOUN, ADJ}, Σ = {the, dog, building, plays, fluffy, good}, and S = SENTENCE. The rules of P are given below.

- SENTENCE $\rightarrow$ the NOUN plays
- NOUN $\rightarrow$ ADJ NOUN
- NOUN $\rightarrow$ dog
- NOUN $\rightarrow$ building
- ADJ $\rightarrow$ fluffy
- ADJ $\rightarrow$ good

We can now start with SENTENCE and apply these rules to construct a "sentence" that is valid in this language. Example sentences and their derivations are given below.

- SENTENCE $\rightarrow$ the NOUN plays $\rightarrow$ the ADJ NOUN plays $\rightarrow$ the good NOUN plays $\rightarrow$ the good dog plays
- SENTENCE $\rightarrow$ the NOUN plays $\rightarrow$ the ADJ NOUN plays $\rightarrow$ the ADJ ADJ NOUN plays $\rightarrow$ the ADJ ADJ ADJ NOUN plays $\rightarrow$ the fluffy ADJ ADJ NOUN plays $\rightarrow$ the fluffy good ADJ NOUN plays $\rightarrow$ the fluffy good fluffy NOUN plays $\rightarrow$ the fluffy good fluffy building plays

Notice that the first example expresses an actual thought whereas the second example, while still syntactically a sentence, is nonsensical. Because it allows meaningless sentences, the grammar described above belongs to the class of *context-free grammars*.

Before we discuss and classify automata in depth, we should first consider what is **not** an automaton. What is an example of something that might perform some kind of calculation, but is not a computer? What about a microwave? Is a microwave a computer? No, it is not. A computer can be programmed in some meaningful, robust way. A microwave contains a microprocessor, which uses *combinational logic* and basic binary inputs to set timers and operate the oven. It cannot be programmed in any meaningful way. What about a calculator? Is a calculator a computer? If we are being formal, the

answer is no, but it depends on what kind of "calculator" we are talking about.

Calculators, such as *counting boards* and the *abacus*, have been around since prehistory. Calculators with four-function capabilities have been around since the invention of Wilhelm Schickard's mechanical calculator in 1642. In the late 19th century, the *arithmometer* and comptometers, two kinds of digital, mechanical calculator, were being used in offices. The Dalton Adding Machine introduced buttons to the calculator in 1902, and pocket mechanical calculators became widespread in 1948 with the *Curta*. None of these are computers.

The difference between a calculator and a computer is that a computer can be programmed and a calculator cannot. What does it mean to be programmable? That is perhaps the central question of automata theory, and we will discuss in this section several levels of "programmability". However, for now, we can certainly say that a simple, four-function electronic calculator is not a computer. It simply uses combinational circuits like full-adders, full-subtractors, multipliers, and dividers to implement its functions, and there is no potential for modifications or user-defined functions.

Surprisingly enough, *special-purpose computers* have also been around for a long time. Early examples include the *Antikythera mechanism* (an Ancient Greek analog computer), Al-Jazari's 1206 water-powered *astronomical clock*, the 1774 *Writer Automaton* (a mechanical doll that could be programmed to write messages), the 1801 *Jacquard loom*, and *slide rules*. Some later mechanical computers were quite powerful, such as *differential analyzers* (special-purpose computers for solving differential equations) and fire control computers. Charles Babbage designed the *Analytical Engine*, a general-purpose computer, in 1837, and Ada Lovelace wrote a program for it, but it was never built. General-purpose computing would not reemerge until the 1940s.

The line between calculator and computer began to blur with the introduction of *programmable calculators* in the 1960s. Many modern high-end calculators are programmable in Turing-complete languages such as TI-BASIC or even C or C++, which officially makes them computers. Once we start implementing *sequential logic* with components like SR latches or D flip-flops, we are storing state, and state is a requirement for computing. Circuits that use sequential logic can be considered automata and, given enough complexity, computers.

We will now discuss four classes of automata, in increasing order of complexity. As they become more complex, automata are able to recognize more powerful and expressive languages. They can be told to execute *instructions* in these languages, and they will be able to accomplish more complex tasks if their language is more expressive. When we reach the last of these four automata, we will be talking about the most powerful model of computation that is feasible to build, the automaton that models modern general-purpose computers. Once we explore this particularly important automaton in depth, we will learn how it can be modified to model more exotic forms of computation.

1.2.1 *Finite-state Machines*

A finite-state machine is a model of computation that can be in exactly one *state* from a finite set of states at any given time. Given its current state and an input, an FSM can *transition* to another state. *Deterministic finite automata* (DFA) are FSMs that transition to at most one state for each input. In contrast, *nondeterministic finite automata* (NFA) can transition to zero or more states for each input. It follows then that a DFA is a special kind of NFA. An FSM may also have a subset of states that are considered *final states*.

When the machine transitions to one of these states, it will finish its work and cease operation.

FSMs can read and understand *regular languages*, which are languages that can be expressed using a *regular expression* or *regex*. A regex consists of constants (which denote sets of strings) and operators (which denote operations on those sets). If an FSM receives an input (such as a regex) at a given state, and its state-transition function maps that state and input to another state, we say that the FSM *matches* that input. That is, it considers that input valid and will perform an action accordingly.

A Quick Summary of Regular Expressions

Given a finite, non-empty input alphabet Σ, there are three constants defined as regexes:

- The *empty set*, $\emptyset$, which denotes the set containing no elements.
- The *empty string*, ε, which denotes the set containing only the empty string "".
- A *literal character* (e.g. the character a denotes the set containing only the character "a").

These constants can be combined and manipulated with operators to create long, complex regular expressions. There are three operators that operate on regexes, which are described below in increasing order of priority. Given regular expressions A and B, the operations include:

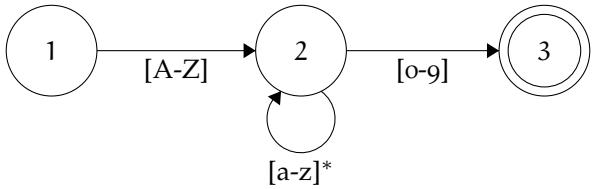
- The *concatenation* of A and B, AB, which denotes the set of strings that can be created by concatenating a string in A with a string in B.
- The *alternation* of A and B, $A|B$, which denotes the union set of A and B.
- The *Kleene star* of A, A^*, which denotes the set of all strings that can be created by concatenating a finite number of strings (including zero strings) from the set A.

Two additional operators are added as "syntactic sugar": $+$ and ?. Whereas the literal character a denotes the set of strings that contain a single a and nothing else, the regex $a+$ denotes the set of strings that contain *at least one* a and nothing else and the regex $a?$ denotes the set of strings that contain *at most one* a and nothing else. Thus, $a+ = aa^*$ and $a? = a|\varepsilon$.

A variety of *metacharacters* are also often used in order to create more concise regexes. Common metacharacters include:

- ^, which matches the starting position in a line of text.
- $, which matches the ending position in a line of text.
- ., the *wildcard*, which matches any character.
- −, which, when placed between two regexes in brackets, denotes a range of characters.
- [^], which matches a single character which is not in the brackets.

Not only is it correct to say that a finite-state machine can understand any regular expression, but also it is true that FSMs and regexes are equivalent. That is, a regex can be written in the form of an FSM and vice versa. Indeed, a regular language "program" can be expressed as a regex, a DFA, or an NFA. For example, the regex [A-Z][a-z]*[0-9] describes any string that starts with a single uppercase letter, is followed by zero or more lowercase letters, and ends with a single digit. This expression can also be represented by the following DFA. Note that state 3 is circled twice to indicate that it is a final state. The automaton will continue to run until it receives a sequence of input that allows it to transition to its final state.



Regular languages and, by extension, finite-state machines are often used in string searching algorithms. FSMs and regular languages are also often used in the lexical analysis (lexing) done by a compiler. That is, they give language designers the power to specify which words are valid in their programming language.

1.2.2 *Pushdown Automata*

Pushdown automata (PDA) are finite-state machines that also have access to a stack. These automata introduce a notion of *history* or *memory*. A PDA can push a symbol onto its stack on every transition, which will provide information in the future about actions in the past. It can also pop or peek at the stack to decide which transition to take next.

Pushdown automata can distinguish syntactically correct sentences from random sequences of valid words, something finite-state automata cannot do because they have no notion of what comes earlier than a given point in a given sentence. For example, a pushdown automaton would be able to tell that "my dog bites red toys" is a valid English sentence and "red my toys dog bites" is not. However, PDA cannot understand semantics. It would also consider "my dog *barks* red toys" a valid English sentence, even though it has no meaning.

Pushdown automata can read *context-free languages*, which are languages that follow a *context-free grammar*. Context-free languages are formal languages whose production rules are of the form $A \to \alpha$ where A is a nonterminal and α is a sequence that may contain terminals and nonterminals. A PDA pops terminals from its stack. When a nonterminal A is at the top of the stack, the PDA can pop it and push the α of some production rule onto the stack. In a sense, the stack of the PDA contains the unprocessed data of the grammar.

Context-free grammars can be written to formalize languages such as the language of matching parentheses or the language of infix algebraic expressions (e.g. (2+4)/7*8). PDA and context-free languages are often used in the syntactic analysis (parsing) done by a compiler. That is, they give language designers the power to specify how valid sentences are structured in their programming language.

1.2.3 *Linear Bounded Automata*

Imagine that, instead of a stack, an automaton could have access to a finite-length list. Linear bounded automata (LBA) have a finite number of *states*, a finite length of *tape*, and a *head* that can read and write symbols on the tape and move left or right along it, one symbol at a time. The length of the tape is a linear function of the length of the input, so we can say that the tape has kn cells, where n is the length of the input instructions and k is a constant.

This machine is similar to the idea of a modern computer. The tape represents a finite amount of memory, and the head is able to read and write to it. LBA can understand *context-sensitive languages*, which are languages that follow a *context-sensitive grammar*. Sophisticated programming languages are context-sensitive, so a linear bounded automaton would be able to read and understand modern software. The kn-length tape acts as a sufficient environment for computationally demanding programming tasks, given a large enough k. Similarly, given enough memory, real computers can also solve very computationally difficult problems.

A context-sensitive language is a language where semantics matter. Using our earlier example, "my dog barks red toys" would not be a valid sentence in a context-sensitive version of English. An LBA then is actually able to differentiate data types and can tell when an operation is defined for one type and undefined for another. LBA and context-sensitive languages are often used in the semantic analysis (type checking) done by a compiler. That is, they give language designers the power to specify whether or not a syntactically correct sentence actually has any real *meaning* in their programming language.

Where can we go from here? LBA are suitable models of real-world computers, and they can process semantic language. What more can we do? Let's try making the tape infinitely long...

1.2.4 *Turing Machines*

A *Turing machine* (TM) not only models real-world computers, but *computation* itself. The concept, which was formalized in 1936 by Alan Turing, generalizes automata and defines the limitations of mechanical computing. It specifies a set of components that are necessary and sufficient for creating a "problem-solving machine" that is capable of solving any problem that can be solved by a machine. Weaker automata can solve *some* of these machine-solvable problems, but only *Turing-complete* automata (those that are functionally equivalent to TMs) have the expressive power to solve *any* of them. The components of a Turing machine are listed below.

- An infinitely long *tape* that is divided into cells, each of which contains a symbol from some finite alphabet. One symbol from this alphabet is considered blank, and the tape is initially filled with these symbols.
- A *head* that can read and write symbols on the tape and move left or right along it, one symbol at a time.
- A *state register* that stores its current state and initially stores its starting state.
- A *finite table of instructions* that, given the machine's current state and tape symbol, tells the machine to do the following sequence of actions:
    1. Erase or write a symbol at the current tape cell (or do nothing)
    2. Move the head one tape cell to the left or right (or do nothing).

3. Depending on the current state and input, transition to a different state (or the same state) or halt computation.

A machine like this need not be a real-world computer. In fact, in his original proof, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Turing refers to a person, whom he calls the "computer", as an example of a Turing machine. If we break down these components into the resources they represent, it becomes clear that many systems could be considered Turing machines.

- The infinitely long tape represents *infinite space for computation* or *infinite memory*.
- The head represents three abilities:
  1. Reading memory,
  2. Writing to memory,
  3. Traversing the memory freely, with no side effects.
- The state register represents the ability to *know what "step" you are on* in the problem-solving process. We will clarify this ambiguous idea in a second, but for now just think of it as having an idea of your progress.
- The finite table of instructions represents a *sequence of commands* or *program* that can be followed unambiguously. The automaton can follow and obey these commands in order without any actual thought.

As long as you have unlimited space to work in, the freedom to read and write symbols anywhere in that space, a list of instructions to follow exactly, and the knowledge of what to do next, you have a Turing machine. We can now visualize the person Turing referred to as a "computer". It is a man with an infinitely large piece of paper, a pen that he writes symbols with, a list of instructions that tells him which symbols to write, where on the paper to move his pen tip, which instruction to perform next, and a knowledge of which instruction he is currently performing. Many systems of logic, such as programming languages, are Turing-complete. Many things not traditionally thought of as "systems of logic" are also Turing-complete.

Turing Machines and Consciousness

In 1950, Turing developed the *Turing test*, a test of a machine's ability to exhibit behavior indistinguishable from that of a human. It involves an interrogator who is tasked with having two separate text conversations and determining which of the two participants is actually a machine. If the interrogator cannot distinguish a difference, the machine has passed the test. Despite its importance to the philosophy of computer science, the test is not a good indicator of whether or not a computer can think. The test says more about how gullible the interrogator is than how conscious the computer is. At the end of the day, the computer is still following its programming.

What about this man whom Turing calls a "computer"? Would he pass the test, given the right programming? Perhaps he would. But that would not make him a *person*. What kind of person is this man if he follows whatever instructions he is given? He is a slave, a person who always obeys. Could we say then that machines are also, in a certain sense of the word, slaves? Perhaps this is where the term *master-slave technology* came from. Regardless, it is important to make a philosophical distinction here about what separates humans from machines

or, more precisely, what distinguishes *thought* from *computation*.

Essentially, this boils down to the question of consciousness. There is much debate about consciousness. Philosophers have proposed a variety of concepts that partially define it such as free will (the ability to choose) or qualia (the raw experience of existence, e.g. sounds, colors, emotions). However, *intentionality* is one characteristic that is generally agreed upon as necessary for conscious thought. Intentionality is the ability of the mind to think *about* something. Computers can *think* things, but they cannot *think about* things. They lack intentionality.

For a Turing machine, the list of instructions is an essential component. It is capable of "thinking" if and only if some one tells it what to think. That effort is more accurately defined by the term "computation". We would also refer to the man's efforts with his pen and infinite paper as computations because they require no thought. If he were nothing more than a Turing machine, the man would not be able to perform mathematics on his own. In fact, he would not be able to do anything without instructions. It is the moment when he does something without being explicitly told to do it that he first displays consciousness. When *unprompted* computation is performed for some *purpose*, we can talk about consciousness.

So calling computers slaves is a bit of an anthropomorphization. A slave is a conscious being who is forced to act like a machine. Computers don't have the prerequisite consciousness. No matter how complicated its architecture or sophisticated its artificial intelligence software, a computer is a *tool* like a microwave or a calculator. Could it ever be more than that? Could a technology ever think? Perhaps. But such a thing would be quite different from a Turing machine or any modern computer.

> *In attempting to construct such machines we should not be irreverently usurping His power of creating souls, any more than we are in the procreation of children: rather we are, in either case, instruments of His will providing mansions for the souls that He creates.*
>
> —Alan Turing (1950),
> in response to a theological objection to artificial consciousness.

A single-tape Turing machine is formally defined as a septuple $(Q, \Gamma, b, \Sigma, \delta, q_0, F)$, where

- Q is a finite, non-empty set of *states*,
- Γ is a finite, non-empty *tape alphabet*,
- $b \in \Gamma$ is the *blank symbol*,
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of symbols that can be written on the tape,

- δ is a partial function $\delta : (Q \setminus F) \times \Gamma \nrightarrow \Gamma \times \{L, R\} \times Q$ called the *transition function*, which inputs the current state and tape symbol and outputs the symbol to write to the tape, the direction to move the head, and the next state to transition to,
- $q_0 \in Q$ is the *starting state*, and
- $F \subseteq Q$ is the set of *final states*. The contents of the tape are *accepted* if the Turing machine halts computation in a state from F.

We should now address what exactly "state" is in regard to computing. Many people associate it with the current instruction being fed to the machine. However, Turing made a distinction between this interpretation of state and the interpretation of state as the computer's "progress" or "state of mind". Turing's *complete configuration* of state includes not only the current instruction, but the current symbol configuration of the entire tape as well and all the instructions yet to be executed. In this way, state is defined by the results of past instructions and the inevitable execution of future instructions.

Now we have defined what a Turing machine is, but it is still not clear why it is considered such a landmark concept in computer science. For example, if real-world computers can be sufficiently modeled by linear bounded automata, why do we instead focus so heavily on Turing machines?

Turing machines are the class of automata that can read *recursively enumerable languages*. A formal language is called recursively enumerable, if it is a *recursively enumerable subset* in the set of all possible words over the alphabet of the language. This essentially means that, for a language of this type, there exists an algorithm that can output a list of every word in the language. Consider what would be required for such an algorithm. How many valid words are there in a given language? Depending on its rules for constructing words, there may be an infinite amount. This is the case for some programming languages. A variable name could theoretically be as long as you want, provided you have enough memory to store it. In order to list all of the words in a recursively enumerable language, you would need infinite memory, which only a Turing machine can provide.

This does not imply that TMs can handle infinite lists of instructions. Rather, they can handle *infinite looping* over a finite set of instructions. A Turing machine can "successfully" run a never-ending program. A real machine would use up all of its memory trying to run such a program, eventually crashing due to a *stack overflow* (an attempt to write data outside of the limits of the memory). A TM would never run out of memory and, given infinite time, could run the program forever.

As previously stated, the Turing machine models not only computers, but *computation*. It abstracts away the physical limitations of computers such as memory constraints, overheating, or hardware failure and asks what the fundamental limits of algorithmic computation are. It makes a statement on which mathematical problems are *decidable*. It is an ideal computer, and, as such, it is not only a model of what real-world computers are today, but a model of what they *could be* in the future, given sufficient advances in hardware.

1.3 The Importance of Turing Machines to Modern Computing

The automata we have discussed so far (finite-state machines, pushdown automata, linear bounded automata, and Turing machines) form a sort of hierarchy of machine capability. The formal grammars and languages that these automata can understand also

constitute a hierarchy that was first described by Noam Chomsky in 1956. The *Chomsky hierarchy*, a classification of the expressiveness of language according to grammatical rules, is summarized below by a table with columns for grammars, the languages those grammars build, and the class of automaton that can understand those languages.

Table 1: The Chomsky Hierarchy

Grammar	Language	Automaton
Type-0	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Linear bounded automaton
Type-2	Context-free	Pushdown automaton
Type-3	Regular	Finite-state machine

As this is a hierarchy, higher-ranking automata are capable of doing anything that lower-ranking automata can. For example, an LBA can do anything that a PDA or FSM can and *more*. To give a linguistic analogy, an LBA would be fluent in all of the languages that the PDA and FSM are fluent in, but would also be fluent in additional languages. What causes this difference in language facility? It is the structure of the automaton's memory.

Let's recap how these four automata handle memory.

- An FSM has *no* memory. It simply has a finite number of states, perhaps represented by a finite list of instructions. It can transition between states, but it has no notion of how it got to any particular state. It records no history.
- A PDA has a *stack* of memory, but this form of memory is restricted. It cannot read or write its memory in any order it likes. It can read the top entry on the stack, but it must delete data to read entries located elsewhere.
- An LBA has a *finite array* of memory. It can read or write this memory in any order it likes, but it has limitations on how much information it can store.
- A TM has an *infinite array* of memory. It can read or write this memory in any order it likes, and it can also store as much as it likes.

It is no coincidence that real-world computers today use array-based memory. Arrays are both an intuitive and Turing-complete way to store information. Now, technically, LBA and TMs use "tape" instead of arrays, but the differences are minimal. In tape memory, cells have relative position, but they are not *labeled*. Modern computers are actually modeled by *register machines*. Register machines are equivalent in expressive power to Turing machines, but their memory is composed of an infinite-length array of uniquely addressed *registers*. Like a tape of cells, an array of registers can be freely accessed. A subset of register machines known as *random-access machines* (RAMs) allow for JUMP instructions (e.g. jump to register #5623) in addition to standard sequential traversal of memory (e.g. move right, move right, move right, etc). This allows computers to accomplish a task with fewer instructions, but the expressive powers of random-access machines and Turing machines are equivalent because both are capable of *eventually* solving the task.

Exploring Exotic Automata

It is worth thinking about automata whose memory is modeled by non-list data structures. For example, a pushdown automaton uses a stack to model its memory and because of this, it is not Turing-complete. What if it used a queue instead? In this case, it would be Turing-complete because it could dequeue items to traverse the memory and then enqueue them to avoid data loss. One could envision this as a Turing machine whose infinite tape ends are glued together to form an infinite loop. The machine can only move in one direction, but it can still access every cell because its tape is circular.

The memory can also be modeled by non-sequential data structures to create some bizarre models of computation. What if the memory of a computer was laid out not as an array, but as an undirected tree? What if it was organized according to an algebraic structure like a monoid or a ring? I'm not even going to pretend that I understand what kind of behavior this would result in. But it is an area of active research.

Other tweaks can be made to the properties of a Turing machine to create new, interesting automata. For example, ω-*automata* (or *stream automata*) are Turing machines that expect an *infinite* sequence of instructions. ω-*automata* never stop running because an infinite sequence of instructions requires an infinite sequence of instruction executions. Because they never terminate, they never move into acceptance (final) states. Rather than a set of final states F, they have a set of *acceptance conditions* Acc.

For ordinary automata, every *run* ρ (i.e. a sequence of n states) ends with a state r_n, and the input is only accepted if this state is final (i.e. $r_n \in F$). For ω-automata, runs are infinite sequences, so they do not end with a state r_n at all. How do we tell if a run ρ should be accepted as a valid set of instructions? We require that $\rho \in \mathrm{Acc}$. That is, if the run is a member of the "set of acceptable runs", it should be accepted. What is the "set of acceptable runs"? That depends on which variant of ω-automaton you are talking about.

The class of ω-automata contains multiple automata with different definitions of Acc. For example, for some subset F (final states) of Q (all states), the *Büchi automaton* accepts those runs ρ, for which there exists a final state that occurs "infinitely often" in ρ. What is a state that is visited "infinitely often"? Given an infinite amount of runtime, some states in Q will be visited an infinite amount of times, and others will not. For example, what if you can transition away from your starting state q_0, but you are not allowed to transition into it. Even given infinite time, q_0 would not be visited infinitely often, and if it were the only state in F, you would never be able to construct a run that would be considered valid by a Büchi automaton. Nondeterministic Büchi automata have applications in "always-on" or "always listening" software, such as those used in highly-autonomous robots or smart speakers like the Amazon Echo, both of which receive instructions based on a never-ending, real-time stream of sensory data.

Wow, that's a lot of abstract mathematics. Why is any of this important for gaining a fundamental understanding of Turing machines or real-world computers? It is important because we can only really grasp their *scope* if we

explore outside of it. Some automata can have properties that are not practical or possible to implement in the real-world. Mathematically, they could have infinite states or continuous alphabets or hyperdimensional memory. But here, once and for all, let's define the scope of computation we will be considering for the rest of this guide.

A modern computer has:

- A **finite** set of states Q. If it were instead infinite, the automaton would have a state for every possible input, and thus would be able to understand any conceivable language. This is far too powerful a machine to build. This would essentially be a universal problem solver.
- A **finite** alphabet Σ. If it were instead infinite, the automaton could have a continuous alphabet. What kind of alphabet's symbols exist on a spectrum? Perhaps you could call *sound* a "language" with a continuous alphabet known as frequency. An automaton with an infinite alphabet would not be digital and would not use bits. It would be an analog computer. Analog computers do exist, and they were popular in the 1950s and 1960s, but nowadays we write software for digital computers. *Fun fact:* analog synthesizers, which are still commonly used in electronic music, are considered a kind of analog computer.
- A transition **function** δ. If it were instead a relation (i.e. inputs are mapped to more than one output), the automaton would be nondeterministic. It is suspected that nondeterministic Turing machines would be able to solve NP-complete problems, which is a computational feat that has not yet been accomplished in a tractable way by a deterministic Turing machine.

While more exotic memory structures are theoretically possible, real-world computers use *arrays* of memory. Since Turing machines can solve any computable problem and data structures are used by computers to solve problems, it follows that data structures can be simulated by Turing machines. Because register machines use array-based memory and are Turing-equivalent, it also follows that **all data structures can be implemented using arrays**. This is a very useful insight, and it will be discussed further in Section 3.

The selection of Turing machines (or register machines) as the model for real-world computers has also influenced decisions in computer architecture. In early computers, the instructions were not integrated into the machine. Code was written on punch cards, which were fed into computers. Eventually, code was stored digitally in programs that were uploaded to the computer's memory. This type of machine is known as a *stored-program computer*.

Where should one store instructions or code in a stored-program computer? Those that have a *Harvard architecture* store their instructions in an *instruction memory* that is separate from the *data memory*. Those that have a *von Neumann architecture* store their instructions and data in the same physical memory, but partition the memory somehow to avoid overwriting the instructions. The von Neumann architecture allows "programs that write programs", such as assemblers, compilers, linkers, and loaders. Modern computers are more von Neumann than they are Harvard because their instructions and

data share an address space, but they are not strictly either. We can describe modern computers as *random-access stored-program* (RASP) machines with *split-cache modified Harvard architectures*.

2 COMPUTABILITY THEORY

Computation is mathematical in nature. Underneath its shiny UI and immersive applications, a computer is simply a machine that can be programmed to do math (and do it *very* quickly in the case of modern computers). With that said, let me pose the central question of computability theory: could a machine (or computer) theoretically solve any given math problem, if given the correct input (or code)? We will consider this question more formally in this section, but it turns out that the answer is no. There exist problems that a computer will never be able to solve with computation, even given infinite resources.

In this section, we will discuss what makes a problem *solvable* and thus within the scope of computational analysis. We will also briefly discuss *unsolvable* problems and how they can be organized into a hierarchy of *unsolvability*. That said, before we discuss either solvable or unsolvable problems, we need to formalize our definition of the term *problem* with regard to the field of computer science.

2.1 The Scope of Mathematical Problem Solving

Since computation is mathematical, it follows that the natural use of computers is to solve *mathematical problems*. A mathematical problem is a problem that is amenable to being represented with mathematics. Are there problems that are not mathematical? That is, are there problems that either *cannot* be formalized with mathematics or *should not* be formalized with mathematics because the result would produce no valuable insight? Of course. You might say that certain "human" problems are not mathematical in nature. For example, can an ethical or moral problem be analyzed or solved *mathematically*? Not necessarily. Mathematics is based in *first-order logic*, and the relationship between logic and ethics is tenuous.

First-order logic is a well-defined system of abstract thought. It consists of *logical objects* (e.g. "for all", "there exists", "such that", "and", "or", "is equal to", etc.) and *non-logical objects* (e.g. constants, variables, functions). The axioms of Zermelo-Fraenkel (ZFC) set theory are written in first-order logic, and ZFC set theory is currently the most accepted *foundation of mathematics*. Any formal mathematical solution, then, must be reducible to a series of inferences made using the ZFC axioms. No other "first principles" may be used.

Consider the trolley problem. Five people are going to be hit by a trolley and killed. If you pull a lever, a single, different person will die instead. Is it ethical to pull the lever? It is fairly intuitive to realize that this problem does not have a mathematical solution. You can make a logical argument one way or the other given some appropriate first principles, but you cannot write a proof for the problem using mathematical axioms. Furthermore, any attempt at quantifying the variables involved in an ethical dilemma will be ill-founded as ethics is outside of the realm of mathematics and first-order logic.

A mathematical problem is a problem that can be fully described using a system of axioms that are expressed in first-order logic. Formally, a mathematical problem is solved by proving whether a statement of first-order logic is true or false. If the *proof* is consistent within its axiomatic system, the statement (or its negation) is considered a *theorem*, a logical consequence of the axioms. This is an important discovery. The return type of a mathematical proof is Boolean.

2.2 Decision Problems and Function Problems

First-order logic is also called *predicate calculus*. It is a system for calculating *predicates*, which are Boolean-valued functions. In mathematics, a predicate maps *propositions* written in first-order logic to Boolean values (i.e. $P : X \to \{\text{TRUE}, \text{FALSE}\}$). In linguistics, a proposition can be written in the form of a *yes-no question* without sacrificing any semantic meaning. For example, evaluating the proposition "The sky is blue." as true or false is the same thing as answering the question "Is the sky blue?" with either yes or no.

More generally, a predicate is a function that receives an input and makes a binary decision about it. In computability theory, the analogous concept is called a *decision problem*, a problem that can be posed as a yes-no question. Decision problems are fundamental to mathematical practice. A proof of a mathematical statement is a decision problem. Additionally, formal verification of a proof involves solving a series of decision problems regarding whether or not each statement in the proof can be reduced to its axioms. The proof is valid if and only if all of its statements are. Decision problems are problems that can be solved by *making a decision* (mapping the question to a yes-no answer).

It is natural to expect that computers, which are tools for doing math, should be able to solve decision problems. That said, computers can handle more than just Boolean-valued functions. They can represent functions of *arbitrary* return type (e.g. integers, words, objects, etc.). This allows them to solve the broader class of *function problems*, problems where a single output is expected for every valid input. Function problems are problems that can be solved by *calculating a function* (mapping the problem input to a solution output).

One important example of a decision problem is whether or not a given function problem is solvable. Let F be a *function problem*, and let D be the *decision problem* of whether or not F is solvable for each of its inputs. Let $f : X \to Y$ be a *function* that solves F, where X is the set of all problem inputs and Y is the set of all problem outputs. Let $G(f)$ denote the set of all ordered pairs $(x, f(x))$ such that $x \in X$. $G(f)$ is known as the *graph* of the function f. Let $d : X \to \{\text{TRUE}, \text{FALSE}\}$ be a decision that solves D. For each $x \in X$, $d(x)$ is TRUE if and only if there exists an ordered pair $(x, f(x))$ in $G(f)$. Otherwise, $d(x)$ is FALSE.

Here's a more concrete example. Let x be any natural number (including 0). F asks "What is $2/x$ for all x?". D asks "Does $2/x$ exist for all x?". In both cases, $f(x) = 2/x$, but F is interested in the *value* of each of the outputs while D is interested in the *existence* of each of the outputs. There is one input for which f has no defined output and that is $x = 0$. In this case, the answer to D is negative, and F is technically unsolvable. That said, we can just move the goalposts a little bit to make F solvable by allowing f to be a partial function: $f : \mathbb{N} \nrightarrow \mathbb{N}$. Partial functions do not necessarily map every member of their domain to a value. Thus, they can be *undefined* for some input values. As long as "undefined" is considered a valid answer to F, F can be considered solvable for its

whole domain.

In computability theory, we are interested in determining whether or not an output exists for each of a problem's inputs. If an output exists, we can be sure it has *a* value, but the particular value is inconsequential. While computers do solve function problems, it is enough to consider only the corresponding decision problem when proving whether a function problem is *solvable* or not.

2.3 "Effective Calculability"

Before the study of computer science, mathematicians sometimes informally described functions as *effectively calculable*. This meant that a correct output could be calculated for any input from the domain of the function, using an *effective method*. A method, in general, is just a "procedure" that does "something". A method is *effective* if it consists of a finite number of instructions and solves a particular problem.

A particular method can be effective for some problems and ineffective for others. For example, typing is a method. If I want to put characters into a text document, typing is an effective method. If I want to plant a flower, typing is not an effective method. However, if I want to plant a flower, building a flower-planting robot and programming it, via typing, to plant a flower *is* an effective method because it actually accomplishes the objective. An effective method that is used to calculate the values of a function is called an *algorithm*.

Much work was done in the 1930s to formalize "effective calculability". The results established *Turing computability* as the correct formalization. This model gives the following fundamental definition: any function whose outputs can be calculated by an algorithm is a *computable function*. Computable functions are precisely those functions that can be computed by a Turing machine. In the mathematical sense, they could also be called *solvable*. A computable or solvable *decision* is predictably called *decidable*. If the functions model mathematical proofs, they could also be called *provable*. Despite a few minor differences, all of these terms get at the same idea.

We will now discuss the history preceding and the circumstances of this foundational work in formalizing "effective calculability" as computability. Much of this work was done between the 1930s and the 1950s, but it built heavily on set-theoretical concepts discovered by German mathematician Georg Cantor in the late 19th century. After a crash course in Cantorian set theory, we will examine computable functions from a variety of angles. Computability is interdisciplinary. The concept *permeates* our reality, and as such its eventual formalization was made possibly by the collective efforts of mathematicians, logicians, linguists, and computer scientists. Today, research on computability is widespread in the study of anything scientific.

After a rigorous study of what is computable, we will conclude this section with a brief exploration of what is not computable. This is a topic that is important to theoretical computer science. By assuming that certain aspects of an abstract machine are *hypercomputational*, we can envision the kind of problems that could be solved if computation were ever to transcend Turing machines.

2.4 The Ballad of Georg Cantor

Georg Cantor (1845-1918) was a German mathematician whose work established *set theory*, a fundamental theory in mathematics. Much of his work was built on generalizations of the set of natural numbers, namely the *ordinal numbers* and the *cardinal numbers*. These sets include both finite and *infinite* quantities, a concept that was considered very controversial in the late 19th century. Despite an extreme amount of backlash, Cantor stood by his set theory and formalized what has been called "the first truly original idea in mathematics since those of the Greeks".

2.4.1 *The First Article on Set Theory*

Cantor began his work in number theory, until his mentor, Leopold Kronecker, suggested that he answer an open question in real analysis: "If a given function can be represented by a trigonometric series, is that representation unique?". A trigonometric series is a series of the form

$$\frac{A_0}{2} + \sum_{n=1}^{\infty} (A_n \cos nx + B_n \sin nx).$$

One common example of a trigonometric series is a Fourier series, which has coefficients A_n and B_n of the form

$$A_n = \frac{1}{\pi} \int_0^{2\pi} f(x) \cos nx \, dx,$$
$$B_n = \frac{1}{\pi} \int_0^{2\pi} f(x) \sin nx \, dx,$$

where f is an integrable function. That said, Cantor's work concerned the general definition of a trigonometric series. He proved that any countable, closed set of natural numbers could encode a trigonometric series that uniquely represents a function. The qualifiers *countable* and *closed* are significant. A countable set is a set whose elements can be counted or enumerated. A closed set is a set that contains its limit points. For a trigonometric series, this means the set of all n must contain a countable number of elements, two of which must be 1 and ∞. 1 is no problem, but what does it mean for a set to contain ∞? And, for example, how can the natural numbers between and including 1 and ∞ be "countable"?

The concept of infinity had been around for a long time by the 1870s, but it had, until this point, been considered a philosophical topic. Aristotle identified a dichotomy between the "potential infinite" and the "actual infinite" in which the former can always have elements added to it while the latter is instead considered "complete". This mindset persisted for around two-thousand years with the majority of scholars believing that "actual infinity" was outside of the purview of mathematics. It was used in mathematical practice, but only in a non-rigorous, conceptual way, as seen in the characterization of limits "tending toward infinity". Actual infinity was considered an "ideal entity", not something that could be studied like finite numbers.

Here, however, Cantor had found a rigorous mathematical object containing infinity as an actual numerical quantity. If a set could represent a countable number of terms in a trigonometric series and was closed on $[1, \infty]$, it could look like $\{1, 2, 3, \ldots, \infty\}$. It was this discovery that caused Cantor to think about the differences between a set like this, and a set like the real numbers, which, in addition to these values, could contain many more. He published his first article on set theory in 1874, stating two theorems

that ushered in a new epoch of mathematical thought.

Cantor's first theorem from his 1874 article states that the set of real algebraic numbers can be put into one-to-one correspondence with the set of positive integers. An *algebraic number* is any complex number that is a root of a non-zero polynomial with rational coefficients. That is, it is any $x \in \mathbb{C}$ that satisfies the following equation:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 = 0,$$

where $a_0, \ldots, a_n \in \mathbb{Q}$, at least one of which must be non-zero. A *real algebraic number* or *algebraic real* is then, logically, any algebraic number with an imaginary part of 0. Complementary to the algebraic numbers are the *transcendental numbers*, the real or complex numbers that are *not* roots of any such polynomials, such as π or e.

Cantor is stating here then that there exists a *one-to-one correspondence* or *bijection* between these algebraic reals and the positive integers (also known as the *natural* or *counting* numbers). An intuitive example of a bijection occurs when you touch the fingertips of your left hand to those of your right hand. Each left fingertip is paired with exactly one right fingertip, and vice versa. No fingertip is left unpaired. With this theorem, Cantor proves that the algebraic reals and the naturals are like the left and right hand. When paired one-to-one, no number from either set is left unpaired.

This discovery is almost unbelievable, and it is totally foreign to anything in finite mathematics. To put this into perspective, consider the fact that the algebraic reals contain the rational numbers. One would think that there would be far more rational numbers than positive integers, but when discussing infinite sets, it turns out that that is not the case. The rationals $\mathbb{Q}$, algebraic reals $\mathbb{A}_{\mathbb{R}}$, integers $\mathbb{Z}$, and naturals $\mathbb{N}$ are all examples of *countably infinite* sets. That is, all of their elements can be put into a *list*. While it may be difficult to see this quality in the rational numbers, it is made easier when you consider that the rational numbers are also called the *measuring* numbers. While there are an infinite number of positive fractional measurements you could make while woodworking or cooking, these measurements can also be listed:

$$\left\{ \underbrace{\frac{0}{1'}}_{0} \; \underbrace{\frac{1}{1'}}_{1} \; \underbrace{\frac{1}{2'} \; \frac{2}{1'}}_{2} \; \underbrace{\frac{1}{3'} \; \frac{2}{3'} \; \frac{3}{1'} \; \frac{3}{2'}}_{3} \; \underbrace{\frac{1}{4'} \; \frac{3}{4'} \; \frac{4}{1'} \; \frac{4}{3'}}_{4} \; \underbrace{\frac{1}{5'} \; \frac{2}{5'} \; \frac{3}{5'} \; \frac{4}{5'} \; \frac{5}{1'} \; \frac{5}{2'} \; \frac{5}{3'} \; \frac{5}{4'}}_{5} \; \underbrace{\cdots}_{n} \right\}$$

Note that each bracket has a number n associated with it. Starting with $n = 1$, the numbers in each bracket begin with $^1/_n$ and the numerator is incremented until you reach $^n/_n$. Then, you continue with $^n/_1$, incrementing the denominator until you reach $^n/_n$ again. Fractions whose values are already in the list are skipped. Negative rational numbers can be added by simply placing a number's complement immediately after itself in the list. Thus, $\mathbb{Q}$ is enumerable and countably infinite.

In his second theorem from his 1874 article, Cantor states that, given any sequence of real numbers $x_1, x_2, x_3, \ldots$ in a closed interval $[a, b]$, there exists a number in $[a, b]$ that is not contained in the given sequence. Essentially, this means that, unlike the sets we just discussed, the set of real numbers $\mathbb{R}$ is *not* enumerable or countably infinite. It cannot be expressed as a list or sequence because there will always be a real number missing. There exists no proper way to "count" the reals. With these two theorems, Cantor discovered that differences can exist between infinite sets. There are *distinct* infinities. He comments on this in the same article:

> *I have found the clear difference between a so-called continuum and a collection like the totality of real algebraic numbers.*

The truth of this "so-called continuum" of the real numbers would continue to evade Cantor for the rest of his life. In the years following his first article of set theory, he made a number of foundational discoveries related to this territory he termed the *transfinite*. He began looking for a bijection between the unit line segment $[0, 1] \in \mathbb{R}$ and the unit square (i.e. a square with sides of length 1). In 1877, in a letter to his friend Richard Dedekind (of *Dedekind cuts* fame), Cantor instead wrote a proof for the existence of a bijection between the unit line and all of the points in an n-dimensional space. Not only could the real numbers between 0 and 1 map *one-to-one* with those in a 1×1 grid, they could map to all of the points in the plane, all of the points in 3-dimensional space, and all of the points in *any* arbitrary dimension. Such was the nature of uncountably infinite sets. Cantor wrote to Dedekind below his proof: "I see it, but I don't believe it!"

2.4.2 *Ordinals and Cardinals*

To aid his exploration into the continuous nature of the real numbers, he formulated the *transfinite arithmetic*, the arithmetic of infinite numbers whose size is somewhere between finite and uncountably infinite. He generalized the natural numbers, introducing two countably infinite sets, those of the *ordinal* and *cardinal* numbers.

Ordinal numbers describe order in a collection. More specifically, they describe the *ordinality* of a number in an ordered set (e.g. 1^{st}, 2^{nd}, 3^{rd}, ...). They include both finite natural numbers such as 1, 2, 3, ... and transfinite numbers such as

$$\omega,\ \omega+1,\ \omega+2,\ \ldots,\ 2\omega,\ 3\omega,\ 4\omega,\ \ldots,\ \omega^2,\ \omega^3,\ \omega^4,\ \ldots,\ \omega^\omega,\ \omega^{\omega^\omega},\ \omega^{\omega^{\omega^\omega}},\ \ldots$$

where ω is the "first infinite ordinal". Note that this sequence of transfinite quantities never ends. Cantor commented on this property, stating that the set of all ordinals Ω cannot have a greatest member. Because Ω is well-ordered, there must exist some number δ that would be greater than all of the numbers in Ω. But δ would belong to Ω because Ω contains all ordinal numbers. This implies that $\delta > \delta$, which is a contradiction. Cantor, a devout Lutheran, called this illusive Ω the Absolute Infinite, a number or set that is bigger than any conceivable quantity, finite or transfinite. This kind of thinking later led to the discovery of a number of *mathematical paradoxes* or *contradictions* in set theory, many of which still exist today.

Cardinal numbers describe the size or *cardinality* of a set (i.e. a set could contain 1 element, 2 elements, 3 elements, ...). Like the ordinals, the cardinals include both finite natural numbers and transfinite numbers, the smallest of which is $\aleph_0$ (aleph-null). $\aleph_0$ is the cardinality of any countably infinite set, such as the natural numbers. In contrast to this, Cantor describes uncountably infinite sets, such as the real numbers, as having cardinality $\aleph_1$ (aleph-one). There are other greater aleph numbers that are studied for their own sake such as $\aleph_\omega$, the first uncountable cardinal number *not* equal to $\aleph_1$. That said, $\aleph_0$ and $\aleph_1$ are sufficient for our purposes. Like the set of all ordinal numbers, the set of all cardinal numbers cannot be completed in any meaningful way and thus it can be described as a set of Absolute Infinite cardinality.

2.4.3 *The Continuum Hypothesis*

For much of his career, Cantor tried to prove the *continuum hypothesis*, which states that there is no set whose cardinality is strictly between that of the integers and the real numbers. This would imply that there is no cardinal number between $\aleph_0$ and $\aleph_1$. If this hypothesis were true, the cardinality of $\mathbb{R}$ ($\aleph_1$) would be equal to the *cardinality of the continuum* $\mathfrak{c}$ (a "continuum" being a set whose numbers "blend" into each other

seamlessly).

To prove that $\mathfrak{c} = \aleph_1$, Cantor sought to relate $\mathfrak{c}$ to $\aleph_0$, and in doing so, he formulated a concept that is very relevant to combinatorics. He defined the *power set operator* $\mathcal{P}$, which maps any set S to its *power set* $\mathcal{P}(S)$, the set of all subsets of S. For example, for a set $S = \{1, 2, 3\}$, $\mathcal{P}(S)$ contains the following sets:

$$\{\} \quad \{1\} \quad \{2\} \quad \{3\} \quad \{1,2\} \quad \{1,3\} \quad \{2,3\} \quad \{1,2,3\}$$

Notice that S has cardinality 3 and that $\mathcal{P}(S)$ has cardinality $2^3 = 8$. For any set S with cardinality κ, $\mathcal{P}(S)$ has cardinality 2^κ, and it turns out that this holds for transfinite cardinals as well. Thus, $\mathcal{P}(\mathbb{Z}) = 2^{\aleph_0}$. This expression is denoted with the character $\beth_1$ (beth-one) according to the following rule: $\beth_{\alpha+1} = 2^{\aleph_\alpha}$. By *Cantor's theorem*, $\beth_1 > \aleph_0$, so we can define $\mathfrak{c} = 2^{\aleph_0} = \beth_1$ and state that $\mathfrak{c} > \aleph_0$. The continuum has a greater cardinality than a discrete set like the integers. The question then becomes: Could $\mathfrak{c}$ be anything less than $\aleph_1$, the cardinality of the real numbers? Or is the set of real numbers the smallest example of a continuum?

With the work of Kurt Gödel in 1940 and of Paul Cohen in 1963, it was established that the continuum hypothesis cannot be proven or disproven. It is *independent* of the axioms of Cantor's set theory. It is also independent of the axioms of the current foundation of mathematics, the *Zermelo-Fraenkel set theory with the axiom of choice* (ZFC set theory). That said, since set theory works regardless of whether or not the continuum hypothesis is true, most mathematicians operate assuming that it *is* true because the set of real numbers does appear to exhibit the behavior we would expect from a "continuum". Thus, independent of any particular set theory, we may assume that, for any transfinite cardinal λ, there is no cardinal κ such that $\lambda < \kappa < 2^\lambda$.

Backlash Against Cantor's Set Theory

Typically, when a proof is submitted, it is either quickly accepted by the mathematical community or quickly shown to be flawed. In the case of Georg Cantor's set theory, controversy loomed for many decades and objections came from many different angles. Most of the grievances came from the *constructivists*, a group that was partially founded by Cantor's mentor, Leopold Kronecker.

Constructivism is a *philosophy of mathematics* that asserts that it is necessary to find or *construct* a mathematical object in order to prove that it exists. Unlike *classical mathematics*, *constructive mathematics* does not adhere to the *law of the excluded middle*, which states that a well-formed proposition must be true or false. This implies that a *proof by contradiction* (a proof of an object's existence founded in disproving the object's non-existence) is invalid. Constructivists took issue with the characterization of *actual infinities* (e.g. the uncountably infinite set $\mathbb{R}$) as legitimate mathematical objects worthy of study. Their mathematical philosophy only permitted the existence of *potential infinities* (e.g. the countably infinite set $\mathbb{N}$). Kronecker did not consider Cantor's original 1874 proof of a difference in cardinality between $\mathbb{N}$ and $\mathbb{R}$ as constructive. He remained staunchly opposed to Cantorian set theory and its hierarchy of the infinite, stating that

God created the natural numbers; all else is the work of man.

There were other mathematical objections to Cantor's findings, such as those directed toward the uncountability of the transcendental numbers. By 1874, only

a handful of transcendentals had been discovered. The constant e was proven to be transcendental the year prior, and π would not be proven to be transcendental until 1882. In stating that the algebraic reals were countable and the reals were not, Cantor implied that *almost all* real numbers were transcendental. That is, if you remove a countable subset ($\mathbb{A}_{\mathbb{R}}$) from an uncountable set ($\mathbb{R}$), you are left with an uncountable subset ($\mathbb{A}^{c}_{\mathbb{R}}$, the *complement* of the algebraic reals, also known as the transcendental reals). Many could not accept that something previously thought to be incredibly rare had instead an uncountably infinite number of examples. Other mathematicians, including Kronecker, refused even to accept Cantor's work as mathematical in nature, believing it to be, at best, philosophical.

In addition to objections from mathematicians, Cantor's set theory received a number of complaints from Christian theologians. Some saw the formalization of an uncountable infinity as a challenge to the uniqueness of the absolute infinity of God. Some associated the transfinite hierarchy with pantheism. Cantor felt strongly that his set theory could exist harmoniously within a Christian framework. Even his notational choices ($\aleph$, ω, and Ω) can be considered an homage to the title of "Alpha and Omega". He associated the Absolute Infinite with God, and felt that transfinite quantities, while infinite, were no challenge to the supremacy of the Lord, averring that

> *... the transfinite species are just as much at the disposal of the intentions of the Creator and His absolute boundless will as are the finite numbers.*

Furthermore, Cantor wrote to numerous prominent theologians, including the Pope, in an attempt to clear up this confusion between the abstract notion of infinity and the actuality of infinity, as he saw it, in God and in Nature:

> *The actual infinite was distinguished by three relations: first, as it is realized in the supreme perfection, in the completely independent, extraworldly existence, in Deo, where I call it absolute infinite or simply absolute; second to the extent that it is represented in the dependent, creatural world; third as it can be conceived in abstracto in thought as a mathematical magnitude, number or ordertype. In the latter two relations, where it obviously reveals itself as limited and capable for further proliferation and hence familiar to the finite, I call it Transfinitum and strongly contrast it with the absolute.*

The onslaught of criticism began to wear Cantor down. Frustrated with the disapproval from his mentor and high-ranking members of his faith and with his inability to solve the continuum hypothesis, he fell into a chronic depression that persisted until his death. He ceased mathematical study for years at a time, writing and lecturing instead on Shakespeare. Nevertheless, some mathematicians, such as David Hilbert, supported his set theory.

Hilbert championed the *transfinitist* philosophy, which claims that infinite sets are legitimate mathematical objects. He believed that Cantor's set theory was the key to axiomatizing all of mathematics, a goal that he would pursue for much of his life. He gave lectures on transfinite arithmetic after Cantor's death, employing an intuitive thought experiment known as *Hilbert's Grand Hotel*. Ultimately, set theory was generally accepted, thanks in large part to Hilbert, who believed in Cantor's work even in the face of a considerable opposition to its fundamental principles.

No one will drive us from the paradise which Cantor created for us.

—David Hilbert

2.4.4 *Cantor's Later Years and Legacy*

In the early 20th century, mathematicians and philosophers had found a variety of paradoxes within Cantor's set theory. The most famous example was *Russell's paradox*, which was discovered by Bertrand Russell in 1901. It posits that, given a set S that is "the set of all sets that are *not* members of themselves", it is unclear whether S contains itself. If it does, it contains a set that *is* a member of itself. If it does not, it does not contain all sets. An alternative, colloquial form of this is the *barber's paradox*: Given a barber who shaves all those, and only those, who do not shave themselves, does the barber shave himself? There is no answer to this question. It cannot be answered within its own axiomatic system.

When he was not hospitalized for disease or depression, Cantor lectured on these paradoxes of his set theory until his retirement in 1913. He lived in poverty, suffering from malnourishment during World War I before succumbing to a heart attack in a sanatorium in 1918.

Georg Cantor's work was revolutionary and had far reaching consequences in every field that made use of mathematics. It drew a line in the sand between *discrete* and *continuous* phenomena. It was Cantor's investigation into and formalization of the infinite that laid the groundwork for ZFC set theory, the current "common language" of mathematics. More than that, however, he shifted the collective perception of the *purpose* of mathematics.

Before this point, mathematics was typically done to understand the natural world. Cantor believed not only that mathematics could describe what he saw around him, but that it could also solve problems that were purely logical, those that existed in the mind. Mathematics was a universe in the abstract, worthy of exploring in the same way that the natural one was. For Cantor, mathematics allowed one to see beyond the limitations of the human senses into worlds of arbitrary dimension, unshackled by physical laws. This philosophy has encouraged the continued development of abstract theories, many of which are later found to have concrete implications for the nature of our reality. Cantor stood firm in opposition to the "oppression and authoritarian close-mindedness" he faced from Kronecker et al. and called for objectivity and truth among his peers, bringing humanity, kicking and screaming, into the modern era of mathematical thought.

The essence of mathematics is in its freedom.

—Georg Cantor

2.5 The Diagonal Argument for Computable Functions

While the story of Georg Cantor and his set theory is interesting in its own right, its relevance to Turing computability and to computer science in general may not be readily apparent. For this reason, I would like to discuss the implications of one final topic related to Cantor, his 1891 *constructive* proof of transfinite cardinality known as the *diagonal argument*. The theorem and its short, elegant proof are recreated below.

> *Theorem:* Given the set T of all infinite sequences of binary digits, if $s_0, s_1, s_2, \ldots, s_n, \ldots$ is any enumeration of elements from T, there exists an element $s \in T$ which does not correspond to any s_n in the enumeration.

> *Proof:* We start with an enumeration of elements from T:

$$
\begin{aligned}
s_0 &= (1, 0, 1, 1, 0, 0, 1, 0, 0, 0, \cdots) \\
s_1 &= (1, 1, 0, 1, 1, 0, 1, 0, 1, 1, \cdots) \\
s_2 &= (0, 0, 0, 1, 0, 0, 0, 1, 1, 1, \cdots) \\
s_3 &= (1, 1, 0, 1, 0, 1, 1, 0, 1, 0, \cdots) \\
s_4 &= (1, 0, 1, 1, 1, 0, 1, 0, 0, 1, \cdots) \\
s_5 &= (0, 0, 0, 1, 0, 0, 1, 1, 0, 0, \cdots) \\
s_6 &= (0, 0, 1, 0, 0, 0, 1, 0, 0, 0, \cdots) \\
s_7 &= (0, 1, 0, 0, 0, 1, 1, 0, 1, 1, \cdots) \\
s_8 &= (1, 1, 1, 1, 1, 0, 1, 0, 0, 1, \cdots) \\
s_9 &= (1, 0, 0, 0, 0, 1, 0, 0, 0, 1, \cdots) \\
&\vdots
\end{aligned}
$$

> We then construct a sequence of binary digits $s^\star$ by making the n^{th} digit of $s^\star$ equal to the negation of the n^{th} digit of the n^{th} sequence given above. Put more simply, we highlight the digits along a diagonal, and flip their values to make $s^\star$:

$$
\begin{aligned}
s_0 &= (\,1\,, 0, 1, 1, 0, 0, 1, 0, 0, 0, \cdots) \\
s_1 &= (1, \,1\,, 0, 1, 1, 0, 1, 0, 1, 1, \cdots) \\
s_2 &= (0, 0, \,0\,, 1, 0, 0, 0, 1, 1, 1, \cdots) \\
s_3 &= (1, 1, 0, \,1\,, 0, 1, 1, 0, 1, 0, \cdots) \\
s_4 &= (1, 0, 1, 1, \,1\,, 0, 1, 0, 0, 1, \cdots) \\
s_5 &= (0, 0, 0, 1, 0, \,0\,, 1, 1, 0, 0, \cdots) \\
s_6 &= (0, 0, 1, 0, 0, 0, \,1\,, 0, 0, 0, \cdots) \\
s_7 &= (0, 1, 0, 0, 0, 1, 1, \,0\,, 1, 1, \cdots) \\
s_8 &= (1, 1, 1, 1, 1, 0, 1, 0, \,0\,, 1, \cdots) \\
s_9 &= (1, 0, 0, 0, 0, 1, 0, 0, 0, \,1\,, \cdots) \\
&\vdots
\end{aligned}
$$

$$s^\star = (\ 0,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ \cdots\ )$$

$s^\star$ must differ from each s_n because their n^{th} digits differ. Thus, $s^\star$ does not belong to the given enumeration of T. Given any enumeration of T, you can always construct an infinite binary sequence that does not appear in it. □

With this information, we can go a step further and say that T is *uncountably infinite*. Note that this result requires that the sequences be countably infinite. This argument has become a common technique in proofs to prove the uncountability of a set whose members are countably infinite. Turing used a diagonal argument to formalize the notion of computability and to prove that the *Entscheidungsproblem*, one of the most important mathematical problems of the 20th century, is undecidable. Next, we will use the diagonal argument to formalize the set of computable functions.

2.5.1 *Uncountably Many Languages*

Consider the alphabet $\{0, 1\}$. With this alphabet, one can construct the set of all binary strings S using the regular expression $\{0, 1\}^*$ (i.e. $\{\varepsilon,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,\ 001,\ \cdots\}$). Strings like these are often called *words* in computability theory, but they could potentially represent very long pieces of text that you might not think of as "words". It may be helpful to think of them instead as syntactically valid building blocks of a particular language. Two other things to note:

1. These binary strings are finite in length, by definition of the Kleene star *. They are the set of finite words that can be constructed using the alphabet $\{0, 1\}$.
2. While this example uses a binary alphabet, the set S could be constructed similarly over a ternary alphabet or an n-ary alphabet where $n \in \mathbb{N}$. Formally, the set of all words over an alphabet Σ is denoted Σ^*. Likewise, the symbols of Σ need not be numbers. An alphabet can contain any symbols, as long as they are distinct and there are a finite number of them.

We can then consider the power set of S,

$$\mathcal{P}(S) = \{\{\varepsilon\},\ \{0\},\ \{1\},\ \{10\},\ \{0,\ 1\},\ \{0,\ 10\},\ \{1,\ 10\},\ \{0,\ 1,\ 10\},\ \cdots\}$$

to be the set of all *languages* that can be created using these binary words. A language L is simply a subset of the set of all possible words over a finite alphabet. Stated formally, $L \subseteq \Sigma^*$. As we discussed previously in the section on automata, languages can conform to a grammar, but for now we will consider them simply as countable sets of words. Words that belong to a given language are called *well-formed words* and those that do not are called *ill-formed words*, but only in relation to that language.

Because S is a countably infinite set and because the power set of a countably infinite set is uncountably infinite (by *Cantor's theorem*), we know that $\mathcal{P}(S)$ is an uncountably infinite set. Each member of $\mathcal{P}(S)$ is a countable set of words, also known as a language. Thus, there exists an *uncountable number* of languages of binary words (and of languages in general). The diagonal argument supports this.

Let each sequence s_n from the above proof represent a language. The "columns of the matrix" each correspond to a binary word from S. Thus, the digits in each sequence

represent whether or not a particular word is well-formed (1) or ill-formed (0) with regard to the language the sequence represents. For example, the language $\{0, 1, 01, 10, 11\}$ would have a 1 in its sequence for the columns corresponding to words $0, 1, 01, 10, 11$ and a 0 everywhere else. We can enumerate the languages as in the proof and flip the values along the diagonal of the enumeration to construct a new language. Thus, the set of all languages of binary words is not enumerable. Rather, there are uncountably many languages.

2.5.2 *Countably Many Turing Machines*

Now, let's require that the sequences be *finite*. Because a Turing machine has a finite number of states and transitions, it can be represented with a finite binary string. Similarly, every natural number can be represented with a finite binary string. Thus, there is a one-to-one correspondence between the set of all Turing machines and the set of all natural numbers. So while there are uncountably many languages, there are only countably many Turing machines to recognize those languages.

We know that Turing machines recognize exactly the recursively enumerable languages R, so we know there are also countably many of these. Taking the complement of R implies that there are also uncountably many *non-recursively enumerable languages* that a Turing machine cannot recognize. In fact, *almost all* languages cannot be recognized by a Turing machine.

Before we define the recursively enumerable languages, we should first define a strict subset of them known as the *recursive languages*. A recursive language can be represented by a *recursive set* of natural numbers. A subset L of the natural numbers is called *recursive* if there exists a total function f such that

$$f(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

Such a function represents the following decision problem: "Let L be a language represented by a set. Given a word w, is w a well-formed word in L?" In the case of a recursive L, this decision problem is called *decidable*.

Similarly, a recursively enumerable language can be represented by a *recursively enumerable set* of natural numbers. A subset L of the natural numbers is called *recursively enumerable* if there exists a partial function f such that

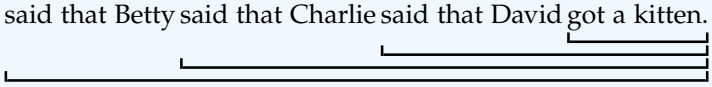
$$f(w) = \begin{cases} 1 & \text{if } w \in L \\ \text{undefined} & \text{otherwise} \end{cases}$$

The decision problem can be phrased in the same way, but, in the case of a recursively enumerable L, it is called *partially decidable* or *semidecidable*. That is, we can decide a word *is* in the language, but we cannot formally decide that it is *not*. In mathematical terms, statements like this can be *proven*, but not *disproven*. If that sounds incredibly unintuitive, that's because it is. This concept will come up later in a discussion of Kurt Gödel's contribution to computability.

Decidability and Semidecidability for Humans and Computers

Recall that recursive languages are also recursively enumerable. That is, their recursive nature is *countable*. This is because recursive languages can construct only *finite* sentences that have an end. On the other hand, a recursively enumerable language can have sentences that expand forever, always ready to accept another clause before its period.

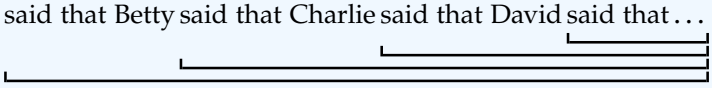
The following sentence is recursive (and, thus, also recursively enumerable):



This sentence is composed of a countable, finite number of building blocks called *clauses* that each contain a *subject* and a *predicate*. In this case, the subjects are the names and their respective predicates are marked by brackets.

All but one of these clauses are *nested* in the predicate of a previous clause. If a sentence can be defined in terms of smaller sentences, it is said to be defined recursively. This sentence qualifies: it contains a full sentence starting with Betty, which contains a full sentence starting with Charlie, which contains a full sentence starting with David. It can also be called recursively enumerable, because the clauses can be counted.

This next "sentence", on the other hand, is recursively enumerable, but it is *not* recursive:



In this case, you can count the potentially infinite number of recursions that occur. However, this "sentence" cannot be called recursive because, due to the absence of a period, it actually contains no full sentences.

How can we understand this concept in the context of computation? We can think about sentences as programs. When someone says a sentence to you, you process information as the sentence is being said. When the sentence finally ends, you have all of the information necessary to understand the thought this person is trying to convey to you. As a result, you gain insight from this person. Similarly, when a programmer executes a program, the computer processes information as it moves forward through the instructions. When the program terminates, the computer has all of the information it needs to compute an answer. It is then able to return a value, which is the kind of insight that is produced by an algorithm.

What if, instead, someone said a sentence to you that never ended? You could listen intently, trying to keep track of everything they've said, but the punchline will never come. You will never understand what they are trying to tell you. Similarly, if a program's instructions never end, a computer will never be able to produce any meaningful answer. This is called an *infinite recursion*, which is a kind of infinite loop.

This conclusion assumes that the computer we are talking about is a real computer in the real world, where time and space are finite. The reason that

a Turing machine is a helpful abstraction is that it has an infinite amount of time and space at its disposal. Theoretically, it could run a program composed of a set of infinitely looping instructions to its "completion". Note the choice of the phrase "a set of infinitely looping instructions" instead of "an infinite set of instructions". The latter is called a *stream*, and it is not Turing-recognizable. ω-automata, however, can recognize it.

Many well-known functions can be defined recursively, such as the factorial function $f(n) = n!$. What is an example of a function that is recursively enumerable, but not recursive? To answer this, I would like to recommend this video, in which a mechanical calculator is instructed to divide a number by zero. The calculator knows the method for division, but this method is only effective in the cases where the denominator is not zero. In the case where the denominator is zero, it attempts to carry out the calculation, faithfully incrementing its count of how many times the denominator fits into the numerator. Of course, zero fits into any number an infinite amount of times, so the calculator will continue to calculate until, as the videographer suggests, it potentially catches fire. The program will never terminate, and a result will never be returned, so we can say that division over any set including zero is only semidecidable.

The problems that Turing machines can solve are either decidable or semidecidable. What, then, is *undecidable*? What kinds of problems do non-recursively enumerable languages describe? Consider the same decision problem from before: "Let L be a language represented by a set. Given a word w, is w a well-formed word in L?" If L is non-recursively enumerable, a Turing machine cannot ever decide this problem because it cannot recognize L. Thus, such decision problems are *undecidable* for all inputs, and any related function problems have no answer.

2.5.3 *Computable Functions and Computable Numbers*

Because all recursive languages are recursively enumerable, a Turing machine can recognize a language from either class. f in both cases is known as a *computable function*, a function whose output can be correctly *computed* by an algorithm performed by a Turing machine. This means that a Turing machine can be considered a *formalization* of the countable set of computable functions:

> A partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ is *computable* if and only if $\exists$ a Turing-recognizable computer program with the following properties:
>
> 1. If $f(\mathbf{x})$ is defined, the program will eventually halt on the input $\mathbf{x}$ with $f(\mathbf{x})$ stored in the tape memory.
> 2. If $f(\mathbf{x})$ is undefined, the program never halts on the input $\mathbf{x}$.

While algorithms are typically written using natural numbers or integers, this is not a requirement. The finite-length k-tuple $\mathbf{x}$ can belong to any A^k where A is a countable set. Likewise, the codomain of f can be any countable set. This generalization allows us to investigate *computable numbers*, the numbers that an algorithm can produce.

Consider a partial function $g : \mathbb{Q}^k \nrightarrow \mathbb{Q}$. Like $\mathbb{N}$, $\mathbb{Q}$ has cardinality $\aleph_0$. Thus, if a program, as defined above, exists, g is computable. If we add a countably infinite set of numbers to $\mathbb{Q}$, we can construct $\mathbb{A}_\mathbb{R}$, the algebraic reals. Because $\mathbb{A}_\mathbb{R}$ also has cardinality $\aleph_0$, a function $g : \mathbb{A}_\mathbb{R}^k \nrightarrow \mathbb{A}_\mathbb{R}$ is also potentially computable. This coheres with our informal notion of computability, as well: The algebraic reals are those real numbers that are the roots of a non-zero polynomial, and an algorithm can compute them by solving their corresponding polynomial equation.

From here, we can add a countably infinite set of transcendental reals to our set of algebraic reals to form the set of *computable numbers*. These are the real numbers that can be computed with an arbitrary level of precision by a finite, terminating algorithm. This implies that a countable number of transcendental reals can be computed. It may be surprising to hear, for example, that the transcendental constant π is computable, but its algorithm is simple. Given a circle, π is the ratio of the circle's circumference to its diameter. These quantities can be stored with an arbitrary level of precision in a TM's countably infinite memory, and instructions can be written to divide one by the other. In general, computable transcendentals can be found by enumerating $S := \mathbb{A}_\mathbb{R}$ and using the diagonal argument to construct a computable real $s \notin S$. Add s to S, apply the argument again, and repeat to add a countably infinite number of computable transcendentals to S.

The set of computable numbers is countably infinite, which means that almost all real numbers are uncomputable. The question naturally arises: Does a number really exist (or have any *worth*) if it is impossible to compute its value? A constructivist would argue that it does not. There are efforts to use the computable numbers instead of the real numbers for all of mathematics, and the resulting theory is called computable analysis. Regardless, one could argue that theoretical computer science is an exploration of the mathematics that can be done within this set.

2.6 Hilbert's Program and Gödel's Incompleteness Theorems

In 1900, David Hilbert gave an address to the International Congress of Mathematicians in Paris. In his speech, he described twenty-three problems that he felt would be some of the most important of the century. While these problems are diverse in subject matter, they are all *deep*. In each of their answers (or lack of answers) lies some fundamental truth about the structure of mathematics itself. They are known as *Hilbert's problems*, and many of them remained unsolved and of great interest today.

This presentation of problems was, in essence, the inception of *Hilbert's program*, a worldwide goal to solve what was known at the time as the *foundational crisis of mathematics*. With the general acceptance of Cantor's set theory came the discovery of many paradoxes and inconsistencies that called into question the *consistency* of mathematics. The creation of ZFC set theory resolved some of these paradoxes, such as Russell's paradox, but not all inconsistencies were resolved. Mathematics was still curiously "broken" in certain ways.

As a proponent of the mathematical philosophy of *formalism*, Hilbert believed that mathematics was not a description of some abstract part of reality, but was rather akin to a game where moves can be made according to a set of rules. He believed that the "games" in mathematics were all played according to some set of *a priori* rules called *axioms*. In the early 20th century, he made it his life's goal to construct such a set of axioms, from which all existing and future theories could be derived in a consistent manner. Mathematicians of the time agreed that the search for such an axiomatization

was direly needed. By this point, no one had been able to prove the consistency of the standard *Peano arithmetic*, which includes axioms such as $x + y = y + x$ and $x \times 0 = 0$. Proving a consistent set of axioms of arithmetic was the second of Hilbert's famous twenty-three problems, preceded only by the problem of solving Cantor's continuum hypothesis.

In order for Hilbert's program to succeed, there would have to exist a formal system with the following properties:

- Completeness, the ability to prove all true mathematical statements
- Consistency, the absence of contradictions
- Conservation, the ability to prove results involving countable quantities without the use of uncountable ones
- Decidability, the existence of an algorithm that could decide the truth or falsity of any mathematical statement

As a professor at the prestigious University of Göttingen, Hilbert gave many lectures on his program, both inside and outside of Germany. At one such lecture given in Bologna in 1928, Kurt Gödel was in attendance. The concepts of mathematical logic discussed at this lecture would inspire Gödel throughout his career, but in only three years he would deal a fatal blow to Hilbert's program.

In 1929, Gödel published his *completeness theorem* of mathematical logic as his doctoral thesis. This was a mathematical proof that elucidated something fundamental about the first-order logic mathematics is based on. Although it is a bit difficult to understand at first, the result is ultimately quite simple, and it supports what mathematicians have implicitly understood about abstract, general proofs since about 300 BCE, when Euclid introduced the *axiomatic method*.

2.7 The Entscheidungsproblem and the Church-Turing Thesis

2.7.1 μ-*recursive Functions*

2.7.2 *The Halting Problem*

2.7.3 *The Untyped* λ*-calculus*

2.8 Computability is Recursion

2.9 Turing Degrees

3 COMPUTATIONAL COMPLEXITY THEORY

Blah, blah, blah...

3.1 Big O Notation

3.2 Complexity Classes

Programming Language Theory

4 ELEMENTS OF PROGRAMMING LANGUAGES

4.1 Syntax

4.2 Type Systems

4.3 Control Structures

4.4 Libraries

4.5 Exceptions

5 A HISTORY OF PROGRAMMING LANGUAGES

6 PROGRAMMING PARADIGMS

6.1 Imperative

6.2 Functional

7 HIGHER-ORDER PROGRAMMING

7.1 Formal Systems of Logic

Abstract Data Types and Data Structures

8 A MOTIVATION FOR OBJECTS

We begin with a definition of a very broad term: a *mathematical object* is an abstract object that can be formally defined. Examples include numbers, matrices, functions, points, lines, shapes, groups, rings, fields, categories, etc. Many mathematical objects are reducible to *sets*, which are mathematical objects that are collections of other distinct mathematical objects.

A *mathematical structure* is a mathematical object that relates to a set and gives that set additional properties. Consider the differences between a set and a *sequence*. A sequence is a collection of mathematical objects, like a set, but it has additional structures that give it additional meaning. Some significant structures a sequence has that a set does not have are *multiplicity*, *metric*, and *order*.

Table 2: Meanings of Multiplicity, Metric, and Order

Structure	Meaning	Example using the sequence $(4, 3, 2, 3)$
Multiplicity	The mathematical object can have duplicate elements.	The element 3 has a multiplicity of 2.
Metric	The mathematical object grants some notion of distance between its elements.	4 is two "steps" away from 2.
Order	Each element in the mathematical object is, in some sense, "less" than or "more" than each of the other elements in the mathematical object.	The "first" element (4) "comes before" the "third" element (2).

If the elements themselves are subject to some kind of *natural order*, the mathematical object would have an additional structure enforcing this order. For example, in a sequence of integers, every element (integer) must have an inherent value that is less than or greater. Integers have a natural order known as numerical order. Similarly, English letters have a natural order known as alphabetical order. An order is considered natural if it is somehow intuitive or widely established.

A mathematical object is also defined by which *operations* can be performed on it and what the result of these operations will be. For example, a sequence S_1 can concatenate with a sequence S_2, and the result is a concatenation of S_1 and S_2. The ability to concatenate with other sequences is a defining characteristic of a sequence. We can say then that mathematical objects are defined by their structures (which enforce relationships between their elements), and their operations (which enforce how their elements can be manipulated). This is similar to our Wikipedia definition of data structures.

Now we must leave the realm of pure mathematics and enter the field of computer science. However, it is important, as we discuss analogous concepts in computer science, that we make a distinction between theory and application. For example, in mathematics, we can use the name of a mathematical object (e.g. "set") in both the abstract and in the concrete sense. In the abstract sense, we use that word to refer to a mathematical object, which is a defined classification or **type** that might describe a concrete object (e.g. "a *set* is a collection of distinct elements with these operations..."). In the concrete sense, we can use that word to refer either to a particular **instance** of that mathematical object or to any instance of that mathematical object that satisfies some additional set of requirements (e.g. "the set $\{1, 2, 3, 4, 5\}$" or "a set of five positive integers less than 10"). This difference between type and instance is crucial.

In computer science, a *data type* is used to describe binary data and its intended use to the compiler. *Primitive* data types have a specified implementation. For example, an `int` is often specified by a language to be a piece of data that is 32 bits in size, and these bits are often mapped one-to-one to an integer value via a *two's complement* operation. An *abstract data type*, on the other hand, is a data type without a specified implementation.

Concrete and Abstract Data Types: Why do we need both?

It is worth asking why this distinction exists. Why would some data types have a standard implementation while other data types must be implemented by the programmer?

Primitive data types, which are concrete, are actually implementations of abstract data types, and the language's compiler has specified these implementations for the sake of simplicity and performance. For example, while most programming languages represent integers using two's complement, they could use other representations like signed magnitude or one's complement. However, computers can do math with two's complement integers faster and with fewer operations than with integers represented in other ways. Thus, a language's compiler may enforce two's complement, preventing the programmer from choosing another representation.

Most programmers agree that this is a good thing. While signed magnitude and one's complement have their advantages, two's complement is far superior for almost all computing tasks. For data types whose choice of representation is not so obvious, it is best to keep things abstract and let the programmer decide which implementation is best for their needs.

An abstract data type is analogous to a mathematical object, which means it is a logical description of the structure of an object and what its operations do. It explicitly does **not** keep track of any sort of *state* (i.e. the values of the elements). It only has to define the structure and operations required for a concrete object to be of that type. But here's the punchline: *operations enforce structure*. An abstract data type can actually be fully defined by the intended effects of its operations. It is a list of functions and descriptions of what those functions should do.

A *data structure* implements the logic that actually produces the desired effects described by each of the abstract data type's functions. It also defines where and how the elements of a concrete instance of a mathematical object should be stored in computer memory, through the use of *fields*. The values of the elements (the *state*) only exist for *instances* of data structures. While a data structure may define the initial state of an instance via a constructor method, it is still the case that only instances of data structures have state.

So we have three terms that span the spectrum of abstract to concrete: abstract data types, data structures, and instances (of data structures). Abstract data types define their structure via a description of intended behavior, data structures define how this intended behavior will be achieved using *algorithms*, and instances exhibit this intended behavior by storing a state that is manipulated by the functions implemented by the data structure. The goal, the plan, and the execution.

Java differentiates abstract data types and data structures by giving them their own *reference types*: `interface` and `class`. The reference type of an instance is created by the user, and the name of this type will be equivalent to the variable name of the class it instantiates. Many object-oriented programming languages have libraries of data structure implementations that programmers can instantiate and the fine-grain details of how these data structures are implemented can vary from language to language. In the next section, we will discuss the most significant data structures in modern software design, and we will do so in a language-independent way, focusing on object-oriented programming in general rather than on any particular OOP language.

9 ABSTRACT DATA TYPES AND THE DATA STRUCTURES THAT IMPLEMENT THEM

In this section, we evaluate data structures from a theoretical standpoint, describing their essential properties and why one may be preferable to another for some task. *There is no single, best way of storing data.* The optimal layout varies depending on the nature of the task. The rules and requirements that define these data structures can be implemented in code, and many programming languages have their own implementations of these structures.

Some operations are common among many different data structures. The time complexity of any of these operations will depend on the choice of data structure. We will discuss how these operations are performed concretely for each data structure listed below, but first we will give an abstract description of these operations.

ACCESS To *access* an item in a data structure is to **provide** an index to denotes the desired element's location relative to the other elements (1st, 2nd, ..., Nth) and to **use** that element to acquire the element's data. The action is only applicable to data structures that have order.

SEARCH To *search* for an element in a data structure is to **provide** some sort of information about the element (like its value) and to **use** that information to acquire the element's data.

INSERT To *insert* an element into a data structure is to **find** a valid location to insert the element, to **allocate** that memory for use by the data structure, and to **store** the

element's data there. Requirements for proper insertion depend on the data structure in question. For example, if the structure requires that its elements be sorted, an element must be inserted into that structure in sorted order. It also may be possible to insert an element into a specified location in the structure, with different locations taking different amounts of time.

The *reassign* operation is similar to insert, but simpler. It consists of finding a valid location and storing data there, no new memory is allocated. Instead, memory that is already a part of the data structure is overwritten.

DELETE To delete an element from a data structure is to **find** where the element is stored in memory and to **deallocate** that section of memory from the total memory controlled by the data structure. The data structure may also have to be reorganized in some way after this deallocation. Like insert, this operation has a runtime that depends heavily on where the element in located within the structure.

Note also that, in many cases, a name can be used to refer to both an abstract data type and a data structure that is an implementation of that type. For this reason, a table of relevant abstract data types is given below.

Table 3: Abstract Data Types

Abstract Data Type	Structure Description
Set	An unordered collection of unique values.
Multiset (or Bag)	An unordered collection of elements that are not necessarily unique.
List	An ordered collection of elements that are not necessarily unique.
Map	A collection of key-value pairs such that the keys are unique.
Graph	A set of elements (called nodes) and a set of pairs of those nodes (called edges). The pairs are unordered for an undirected graph or ordered for a directed graph.
Tree	A directed, acyclic graph in which all nodes have an in-degree of 1 (except the root node, which has an in-degree of 0).
Stack	A LIFO-ordered collection of elements that are not necessarily unique.
Queue	A FIFO-ordered collection of elements that are not necessarily unique.
Priority Queue	A priority-ordered collection of elements that are not necessarily unique.
Double-ended Queue	A queue that can add/remove elements from both ends.
Double-ended Priority Queue	A priority queue that can add/remove elements from both ends.

Each data structure described below could implement one or more of these abstract data types. However, each name used below refers to a data structure of one of these types, not to the type itself, unless specified otherwise.

9.1 Lists

A *list* is an abstract data type that represents an ordered collection of elements that are not necessarily unique. It is typically implemented with either static or dynamic arrays or with linked lists. The major difference between these data structures has to do with how the data is stored in memory (i.e. whether data is stored contiguously or not and whether the amount of allocated memory is fixed or not).

9.1.1 *Arrays*

*** Talk about how strings are often implemented with arrays.

An *array* is a collection of elements that are stored contiguously in memory. Each element in the array is assigned an index which describes its relative position in the array.

The elements should take up the same amount of space in memory. This is required to allow indexing to function correctly. To find an element, you must find the beginning of that element's data in memory and then fetch a number of bits that corresponds to the element's size. To find the element with index `i`, you would start at the memory address where the array begins, skip forward `i`×`elementSize` bits, and then fetch `elementSize` bits. In addition to elements taking up the same amount of space, they should also have the same type. This allows the bits to be interpreted in an unambiguous way.

Arrays can be *static* or *dynamic*. A static array takes up a fixed amount of size in memory while a dynamic array can be resized. The difficulty with increasing the size of an array is that an array must remain contiguous, but the memory just ahead of the end of the array could contain important data that cannot be erased. For this reason, dynamic arrays are often implemented as static arrays that are destroyed and recreated with a larger size elsewhere in memory when they exceed their previous size.

Table 4: Dynamic Array Time Complexity

Operation	Time	Reasoning
Access	$O(1)$	Indicies allow for direct access
Search	$O(n)$	Have to check elements linearly
Insert (beginning)	$O(n)$	Have to shift all elements one to the right
Insert (middle)	$O(n)$	Same as above in the worst case
Insert (end)	$O(1)$ amortized	If there is allocated space at the end, it takes constant time. If not, you may have to copy all elements to a new, bigger array.
Delete (beginning)	$O(n)$	Have to shift all elements one to the left
Delete (middle)	$O(n)$	Same as above in the worst case
Delete (end)	$O(1)$	Free the memory at the end of the array

Worst-Case Summary		
Access = $O(1)$	**Search =** $O(n)$	**Insert/Delete =** $O(n)$

A static array has the same search properties, but it cannot insert or delete items, because those operations involve changing the size of the array.

9.1.2 *Linked Lists*

A *linked list* is a collection of *nodes* that are not stored contiguously in memory. A node contains some data and a pointer to the next node in the sequence. To *point* a node A to another node B means to assign A's pointer the value of B's location in memory. The first node is called the *head* of the linked list and the last node is called the *tail*. The tail has no next node, so its pointer points to null.

The data contained in the nodes of a linked list do not need to have the same type or size. Because the nodes are stored non-contiguously and are accessed via pointers, there is no need for nodes to be of the same size.

Linked lists allow for insertion and deletion of elements without restructuring the entire data structure. This means that linked lists are dynamic. To insert an element X between elements P and N, traverse the list to find P, save P's pointer to N as temp, point P to X, and point X to temp. Deleting is a similar process: traverse the list to find P and point it to N.

Because you have to traverse the list to insert or delete, it is common practice to keep track of where the tail is located in memory. This allows for constant time insertion of elements at the end of the list. For constant time deletion of the tail, you would also have to cache the penultimate node, Pen, in order to point it to null and make it the new tail. However, keeping track of Pen is $O(n)$ for a singly linked list because there is no way of directly finding the new Pen after deleting the tail once.

A linked list can be singly or doubly linked. A doubly linked list has nodes with pointers to both the previous and next nodes in the sequence. This allows for traversal

of the list in both directions. It also allows for constant time deletion of the tail, as long as the tail is cached.

Table 5: Linked List Time Complexity

Operation	Time	Reasoning
Access	$O(n)$	Linked lists do not really have indicies, but you can iterate through a certain amount of pointers, starting at the head
Search	$O(n)$	Have to check elements linearly
Insert (beginning)	$O(1)$	Point the new node to the head
Insert (middle)	$O(i)$	Search for the previous node at index i-1, then reassign pointers
Insert (end)	$O(1)$ with caching, $O(n)$ otherwise	If you know where the tail is stored, you can just point it to the new node. If not, you must first traverse the entire list to find the tail, which is an $O(n)$ operation.
Delete (beginning)	$O(1)$	Point head to null
Delete (middle)	$O(i)$	Search for node at index i, then reassign pointers
Delete (end)	$O(1)$ with caching and double links, $O(n)$ otherwise	To delete the tail, you need to know where the penultimate node, Pen, is. If you cache the tail and have double links, you can always find Pen. Otherwise, you must traverse the list to Pen, which is $O(n)$.

Worst-Case Summary (singly linked, no caching)		
Access = $O(n)$	**Search =** $O(n)$	**Insert/Delete =** $O(n)$

Worst-Case Summary (doubly linked, cached tail)		
Access = $O(n)$	**Search =** $O(n)$	**Insert/Delete =** $O(1)$

9.1.3 *Skip Lists*

Blah

9.2 Stacks

A *stack* is a collection of elements with two operations: *push* and *pop*. To push an element onto a stack is to add it to the stack. To pop an element from the stack is to remove its most recently added element and return it. *Peek* is an operation that is sometimes implemented for convenience. It allows access to the most recently added element without removing it from the stack. The most recently added element is located at the

top of the stack, and the least recently added element is located at the *bottom* of the stack.

A stack is similar to a linked list, but its operations are stricter. It can only insert and delete elements at the head of the list (top of the stack), and searching for or accessing an element located somewhere other than the head requires removing elements from head to tail until the desired element is the head. A stack can be implemented using an array or a linked list, so whether or not it is contiguous depends on the implementation details.

Stacks Implemented at the Hardware Level

Stacks are also used in the architecture of computer memory. Classically, the bottom of the stack will be placed at a high address in memory and a stack pointer will be assigned its location. When data is pushed onto this stack, the stack will grow downward to lower addresses in memory, and the stack pointer will keep track of the top of the stack (lower addresses). When data is popped, the stack pointer will move accordingly toward the bottom (higher addresses).

Local function variables are stored on the stack. If a call to another function is made, that function's local variables will be stored on the stack in a *frame*. From top to bottom (low to high address), the frame of a callee function typically stores local variables, the return address to the code being executed by the caller function, and then parameters of the callee function. The *frame pointer* marks the location of the return address in the frame. This implements the concept of *scope* in computing.

Table 6: Stack Time Complexity

Operation	Time	Reasoning
Access	$O(n)$	To access the bottom element, you must pop every other element
Search	$O(n)$	If the element you are searching for is at the bottom, you must pop every other element
Push	$O(1)$	Make the new node the head of the list
Pop	$O(1)$	Remove and return the head and make the next element the new head
Peek	$O(1)$	Return the head

9.3 Queues

A queue is a collection of elements with two operations: *enqueue* and *dequeue*. To enqueue an element is to add it to the queue. To dequeue an element is to remove its least recently added element and return it. As with the stack, *peek* is often implemented for

convenience, and it returns the least recently added element. The least recently added element is located at the *front* of the queue, and the most recently added element is located at the *back* of the queue.

While a queue can be implemented using an array, it is more commonly implemented using either a singly or doubly linked list or using a dynamic array variant called an "array deque". If implemented using a singly linked list, it must insert elements at the tail and remove them at the head. A queue may or may not be contiguous, depending on the implementation.

Table 7: Queue Time Complexity

Operation	Time	Reasoning
Access	$O(n)$	To access the back element, you must dequeue every other element
Search	$O(n)$	If the element you are searching for is at the back, you must dequeue every other element
Enqueue	$O(1)$	Add the new node to the tail of the list
Dequeue	$O(1)$	Remove and return the head and make the next element the new head
Peek	$O(1)$	Return the head

9.4 Deques

Blah

9.5 Priority Queues

Blah

9.6 Graphs

A graph is a set of vertices (nodes) and edges between those vertices. Graphs are either undirected or directed, which means their edges are unidirectional or bidirectional, respectively. It is usually implemented using either an *adjacency list* or an *adjacency matrix*.

With an adjacency list, the vertices are stored as objects, and each vertex stores a list of its adjacent vertices. The edges could also be stored as objects, in which case, each vertex would store a list of its incident edges, and each edge would store a list of its incident vertices. This implementation could, for example, sufficiently represent a *hypergraph*, which is a generalization of a graph in which an edge can join any number of vertices.

With an adjacency matrix, the rows would represent the source vertices and the columns would represent the destination vertices. The matrix simply stores how many edges are incident to both the source vertex and destination vertex. The data pertaining to the vertices and edges is stored outside of the matrix. The adjacency matrix for an

undirected graph must be symmetrical, whereas this is not the case for directed graphs.

Adjacency lists are better at representing *sparse* graphs (graphs with few edges) while adjacency matrices are better at representing *dense* graphs (graphs with close to the maximum number of edges).

Table 8: Adjacency List Time Complexity

Operation	Time	Reasoning
Insert vertex	$O(1)$	Store vertex in hash table, map it to adjacent vertices
Insert edge	$O(1)$	Add an adjacent vertex
Remove vertex	$O(\|V\|)$	Visit all adjacent vertices of the given vertex, remove the given vertex from their adjacency lists, remove given vertex from map
Remove edge	$O(1)$	Remove destination vertex from source vertex's list of adjacent vertices
Check adjacency	$O(\|V\|)$	In the worst case, a vertex could have an adjacency list containing all vertices in the graph
Space complexity: $O(\|V\| + \|E\|)$		

Table 9: Adjacency Matrix Time Complexity

Operation	Time	Reasoning
Insert vertex	$O(\|V^2\|)$	Matrix must be resized
Insert edge	$O(1)$	Increment value in matrix
Remove vertex	$O(\|V^2\|)$	Matrix must be resized
Remove edge	$O(1)$	Decrement value in matrix
Check adjacency	$O(1)$	Check if value in matrix is greater than zero
Space complexity: $O(\|V\|^2)$		

Talk about graph coloring and MSTs and stuff

9.7 Trees

A *tree* is a collection of nodes that contain data and pointers to child nodes. Each child can only have one parent node. There are many descriptors that can be applied to trees. Those that describe some of the most useful tree implementations are listed below.

Table 10: Tree Descriptors

Descriptor	Meaning
Binary	Each node in the tree has at most 2 children
Balanced	The left and right subtrees of each node differ in height by no more than one
Ordered (or sorted)	The nodes are sorted in some way, such that they can be searched in $O(\log n)$ time
Complete*	Every level of the tree has the maximum amount of nodes possible, except for, perhaps, the last level
Full*	Each node in the tree has either zero or two children
Perfect*	Each non-leaf node has two children, and all leaves have the same depth

* These terms are not standardized, but they are often defined this way.

We will now discuss a variety of tree-based abstract data types and data structures that have proven to be very useful in software design. We will cover the binary search tree, the binary heap, and the trie, as well the preferred implementations for each.

9.7.1 *Binary Search Trees*

A *binary search tree* (BST) refers to an ordered binary tree that satisfies the *binary search property*, which states that each parent node must be greater than or equal to the nodes in its left subtree and less than the nodes in its right subtree. This is a sorted order that is imposed on the tree to give it $O(\log n)$ search, insert, and delete operations, as long as the tree is also balanced.

Inserting a node, N, into a binary search tree involves searching for it in the tree until you arrive at a leaf and then making N a child of that leaf. Deleting a node, N, is more complicated. If N has no children, simply remove it. If it has one child, replace it with that child. If it has two children, copy the data of C, which can be either N's in-order predecessor or in-order successor, to N. If C is a leaf, remove it. Otherwise, C has one child. Replace C with that child.

Table 11: Binary Search Tree Time Complexity

Operation	Time	Reasoning
Search	$O(\log n)$	Perform a binary search
Insert	$O(\log n)$	Perform a binary search for the closest node and insert the given node as its leaf
Delete	$O(\log n)$	Perform a binary search to find the given node, delete it, and rearrange the tree

As nodes are inserted into and deleted from a binary search tree, the tree may become unbalanced. For this reason, *self-balancing BSTs*, such as *AVL trees* and *red-black*

trees, are often the preferred BST implementations because a BST's search, insert, and delete operations are only $O(\log n)$ if the tree remains balanced.

AVL TREES

In an AVL tree, each node, N, stores the heights of its left and right subtrees, L and R. The difference between these heights (L.`height` - R.`height`) is called N's *balance*. If its balance is less than -1 or greater than 1, N is unbalanced and must be fixed using *tree rotation*. In practice, if a node becomes unbalanced after an insertion, its balance will be either 2 (L is taller) or -2 (R is taller). An AVL tree is only balanced when all of its nodes are balanced.

There are two types of rotations, left and right, and they are inverse operations of each other. A rotation is a way to move a parent (A) down the tree while moving its child (B) up and preserving the order of the tree. Regardless of the direction, this results in B abandoning its child C to adopt A as its child and A adopting C as its child. During a right rotation, B starts as A's left child, C starts as B's right child, A becomes B's right child, and C becomes A's left child. During a left rotation, B starts as A's right child, C starts as B's left child, A becomes B's left child, and C becomes A's right child. The rotation operation takes the parent node, A, as input.

If node N is unbalanced, rotations must be performed to balance its subtrees. The pseudocode for this algorithm is given below.

Algorithm 1: Balancing a node in an AVL Tree

```
Data: A tree rooted at N
begin
    if N is not balanced then
        L ⟵ N.left;
        R ⟵ N.right;
        if L is taller than R then
            if L's extra node is on the right then
                leftRotation(L);
            end
            rightRotation(N);
        end
        else if R is taller than L then
            if R's extra node is on the left then
                rightRotation(R);
            end
            leftRotation(N);
        end
    end
end
```

When a node is added to the AVL tree with a BST insert operation, this algorithm is applied to all of its ancestors as the recursive call stack moves up the tree. This ensures that the tree is balanced. Deletion is handled similarly. A BST delete operation is performed on a node, and then the balancing algorithm is applied to all of its ancestors recursively.

RED-BLACK TREES

A red-black tree is another kind of self-balancing binary search tree in which each node stores an extra bit that colors the node red or black. A red-black tree maintains its balance such that search, insert, and delete operations remain $O(\log n)$, but its balances requirements are looser. Whereas an AVL tree can only have a max difference in height of 1 between subtrees, a red-black tree can have subtrees whose heights differ by up to a factor of two.

In addition to the standard properties of a binary search tree, a red-black tree has five properties that enforce its balance:

1. Each node is either red or black.
2. The root is black.
3. All leaves, which are null nodes, are black.
4. Every red node must have two black children.
5. Every path from a given node to a descendant leaf must have the same number of black nodes.

Consider two paths from a node to its leaves, one with the minimum possible number of red nodes and one with the maximum possible number of red nodes. Each path must have the same number of black nodes, b. The minimum number of red nodes is zero, so the first path has b nodes. Because red nodes must have black children and because each path must have b black nodes, the second path has a maximum number of b red parents and $2b$ nodes total. Thus, two paths from a given node to its leaves differ in height by at most a factor of 2, which preserves the efficiency of the BST's operations.

Inserting and deleting nodes can be complicated with red-black trees because all five red-black properties must be preserved. The operations required to preserve them depend on the colors of nodes related to the node being inserted or deleted.

Red-Black Insertion

A new node, N, replaces a leaf and is *always* initially colored red. It is also given two black, null leaves as children. Because we are inserting a red node, Property 5 is not at risk of being violated, but Property 4 is. We must rebalance the tree in cases where the insertion of N causes a red node to have a red child. In these cases, N's parent, P, must be red. We can also assume that N's grandparent, G, is black, because its child, P, is red. However, G's other child (N's uncle), U, could be red or black.

Case 1: N is the root of the tree.

Just change N's color to black to comply with Property 2.

Case 2: P is black.

Property 4 cannot be violated, so no rebalancing is required.

Case 3: P is red and U is red.

Property 4 is violated because P is red and its child, N, is also red. This can be fixed by flipping G to red and flipping P and U to black. However, G may now violate Property 2 (if it is the root) or Property 4 (if its parent is red). For this reason, the rebalancing function (which was called on N) should be called recursively on G whenever a Case 3 situation occurs.

Case 4: P is red and U is black.

Property 4 is violated because P is red and its child, N, is also red. In this case, it is significant whether or not N and P are left or right children of their respective parents. This leads to four subcases that are all handled with tree rotations, similar to AVL insertion.

Case 4.A: N and P are left children.

Perform a right rotation on G and flip the colors of P and G.

Case 4.B: N is a right child, and P is a left child.

Perform a left rotation on P to create a Case 4.A subtree rooted at G. Perform the Case 4.A steps.

Case 4.C: N and P are right children.

Perform a left rotation on G and flip the colors of P and G.

Case 4.D: N is a left child, and P is a right child.

Perform a right rotation on P to create a Case 4.C subtree rooted at G. Perform the Case 4.C steps.

The logic of red-black balancing after insertion can be summarized by the following algorithms:

Algorithm 2: Red-Black Balancing After Insertion

```
void rbInsertBalance(N):
    Data: A red node, N, that was just inserted into the red-black tree
    begin
        P ⟵ parent(N);
        G ⟵ grandparent(N);
        U ⟵ uncle(N);
        if P == NULL then
            case1(N);
        end
        else if P is black then
            case2(N);
        end
        else if P is red and U is red then
            case3(N,P,U,G);
        end
        else   \\ P is red and U is black
            case4(N,P,G);
        end
    end
```

Algorithm 3: Red-Black Balancing Cases

```
void case1(N):
    begin
        Color N black;
    end
void case2(N):
    begin
        return;  // tree is already balanced
    end
void case3(N,P,U,G):
    begin
        Color P black;
        Color U black;
        Color G red;
        rbInsertBalance(G);
    end
void case4(N,P,G):
    begin
        if P is a left child then
            if N is a right child then
                leftRotation(P);
            end
            case4A(P,G);
        end
        else
            if N is a left child then
                rightRotation(P);
            end
            case4C(P,G);
        end
    end
void case4A(P,G):
    begin
        rightRotation(G);
        Color P black;
        Color G red;
    end
void case4C(P,G):
    begin
        leftRotation(G);
        Color P black;
        Color G red;
    end
```

The process for balancing after deleting is similar, but it is more complicated and involves more cases. As such, it is not worth the space required to expound on it here. Refer instead to the description on Wikipedia's red-black tree page.

9.7.2 *Binary Heaps*

A *binary heap* is a complete ordered binary tree that satisfies the *heap property*, which states that each parent node is either greater than or equal to its children (in the case of a *max-heap*) or less than or equal to its children (in the case of a *min-heap*).

Binary heaps are often implemented using static or dynamic arrays. The root of the heap is stored at index 0. Each of the other nodes is stored at an index $i > 0$ such that the locations of its parent, left child, and right child can be calculated according to the following expressions.

Table 12: Relative Indices for a Heap Represented by An Array

Node	Index
Current Node	i
Parent	$(i-1)/2$
Left Child	$(2 \times i) + 1$
Right Child	$(2 \times i) + 2$

Assuming an array implementation, we can insert a new node in the heap by adding it to the end of the array and resolving any violations of the heap property that occur, if any. We can also delete any node in the heap (including the root) by removing it, replacing it with the last node in the array, and resolving any violations of the heap property that occur, if any. A heap is an optimal choice for implementing a priority queue, which has fundamental operations such as insert, find min/max, and delete min/max.

Table 13: Binary Heap Time Complexity

Operation	Time	Reasoning
Insert	$O(\log n)$	Insert node at end of array and "bubble up"
Find Min/Max	$O(1)$	The min/max (depends on heap order) is always stored at index 0
Delete Min/Max	$O(\log n)$	Replace root with last node and "bubble down"

9.7.3 *Tries*

A *trie* is an ordered tree that is typically not binary. It is similar to a map, but the keys (which are usually strings) are not necessarily associated with every node. Instead, a node may store a prefix of a key (such as the first character of a key) and have an incident edge that points to a key or to another prefix that adds a single character to the original prefix. The root node stores an empty string.

This allows for the compact storage and convenient search of strings that share prefixes. This would be useful for storing a dictionary of English words and recommending valid words given a prefix. For this reason, it is often used for autocompleting words and storing new, custom words.

Table 14: Trie Time Complexity

Operation	Time	Reasoning
Search	$O(n)$	Traverse the trie character-by-character until the full n-length string is found or you hit a leaf
Insert	$O(n)$	Add new prefix nodes to the trie character-by-character until there is an n-length path leading from the root to the given word
Delete	$O(n)$	Traverse the trie toward the given string and delete the first node you encounter with an out-degree of 1.

9.8 Maps

A *map*, also known as an *associative array*, is a collection of key-value pairs, such that keys are unique. A key can be used to search the map to find its corresponding value. Maps are typically implemented with a hash table (to make a hash map) or a tree (to make a tree map). Whether or not a map has order depending on its implementation. Hash maps are not ordered while tree maps are ordered.

9.8.1 *Hash Maps*

A hash table uses a hash function to map keys to *buckets* in an array. It does this by hashing a key and reducing that hash to an index in the array, using a modulo operator (`index = hash % array_size`). It must also handle *collisions*, which occur when two or more keys map to the same bucket. The two common ways of handling this are *separate chaining* and *open addressing*.

SEPARATE CHAINING

In separate chaining, buckets hold pointers to linked lists, which hold the key-value pairs in nodes. A bucket without a collision will point to a single node, whereas a bucket with a collision will point to a chain of nodes. The buckets could also instead hold the heads of the linked lists, which would decrease the number of pointer traversals by 1, but would also increase the size of the buckets, including the empty ones, if the values take up more space than a pointer.

OPEN ADDRESSING

In open addressing, all key-value pairs are stored in the bucket array. When a new pair has to be inserted, the bucket corresponding to its hashed key (*preferred bucket*) is checked for occupancy. If there is a collision, more buckets are checked in a *probe sequence*. The most common probe sequence is *linear probing*, which means you check buckets separated by a fixed interval, which is usually 1 (bucket x, bucket x+1, bucket x+2, ...). When an unoccupied bucket is found, the pair is stored there. When a key is used to access a value, the key is compared to the key stored in the preferred bucket. If it matches, the value in that bucket is returned. Otherwise, the process is repeated with other buckets according to the probe sequence and will return either when the keys match (item found) or when the bucket is empty (item not found).

DYNAMIC RESIZING

A hash table that is quite full is slower than one that is quite empty, regardless of the style of collision resolution. If table that uses separate chaining is quite full, it is more

likely that future items will have to be chained to existing items, which leads to longer search times. If a table that uses open addressing is quite full, a future item whose preferred bucket is filled will spend longer looking for an empty bucket, and it will take longer to search for it as well.

The load factor of a hash table is the ratio of the number of keys stored in the table to the number of buckets in the table. When the load factor exceeds some limit (0.75 is commonly used), the hash table will be *rehashed*, which involves creating a new hash table and remapping all of the elements to it. There are other techniques that allow for incremental resizing instead of this all-at-once method, which can be useful for highly available systems.

Table 15: Hash Map Time Complexity

Operation	Time	Reasoning
Access	N/A	Hash maps have no total order
Search	O(1)*	The hash tells you where the data is located
Insert	O(1)*	The hash tells you where to put the data
Delete	O(1)*	The hash tells you which bucket to empty

* This time complexity assumes a good hash function with minimal collisions. A very bad hash function with result in an $O(n)$ time complexity.

9.8.2 *Tree Maps*

A perfect hash function results in an $O(1)$ search time. A particularly bad hash function could cause a hash table with separate chaining to put all key-value pairs into the same bucket, which would result in an $O(n)$ search time. A tree implementation lands in between these scenario with an $O(\log n)$ search time. It can be visualized as a hash table with one bucket which contains the root of a tree instead of the head of a linked list.

Maps like this are often implemented using self-balancing binary search trees like AVL trees or red-black trees. While they are less time-efficient than maps that have good hash functions, they are sorted by key, and thus allow for fast enumeration of items in key order. However, their search algorithms become complicated with the presence of collisions.

Table 16: Tree Map Time Complexity

Operation	Time	Reasoning
Access	N/A	Tree maps have no single *unambiguous* total order
Search	$O(\log n)$	Traverse the search tree
Insert	$O(\log n)$	Traverse the search tree and insert the value
Delete	$O(\log n)$	Traverse the search tree and delete the value, reorganizing the tree if necessary

9.9 Sets

A set is an unordered collection of unique elements. It is typically implemented in a similar way to a map. A hash table is used for unsorted sets to achieve $O(1)$ search/insert/delete. A self-balancing binary search tree is used for sorted sets to achieve $O(\log n)$ search/insert/delete and fast enumeration in sorted order.

A set can also be implemented using a map. In this case, the keys are the elements and the values are flags, such as 1 or `true`.

9.10 Multisets (or Bags)

Blah

9.11 Monoids

Blah

Algorithms

How can you classify algorithms? By problem, time complexity, paradigm?

BRUTE-FORCE description

GREEDY description

DIVIDE AND CONQUER description

DYNAMIC PROGRAMMING description

10 SEARCHING

While data structures are *used* for storing data, they are only *useful* if their data can be quickly retrieved. Given a collection of data, how would you find a particular item in it? That is, given a key (something that identifies the item), how would you find the value (the actual item)? Search algorithms are also used to determine if a data structure *contains* a certain value. In this case, instead of returning the value you searched for, you would simply return whether or not the value was found.

There are many different kinds of search algorithms and one may be preferable to another depending on the data structure, how the data is ordered, and the key you are given. For data structures like hash maps, searching takes constant time. That is, given a key, you can perform a constant time operation (i.e. hashing) to find the value. Data structures with non-constant search times are more interesting and will be the subjects of this section. Trees and graphs in particular have many possible search algorithms that are useful in different scenarios.

10.1 Depth-First Search

Depth-first search (DFS) is a method of traversing trees and graphs. While it is often implemented for the purpose of searching, it is more generally a method of visiting all of the nodes in a tree or graph. It involves starting at a root node and exploring each path as far as possible before searching another path. At each node, you can perform some kind of action (such as comparing the node's data to your key) or move on to another node.

For binary trees, each node has two paths. At any node, you can recursively traverse its left path (L), recursively traverse its right path (R), or process the node itself (N). The order in which these actions are performed determine the order in which the tree is searched. Below are some different tree traversals (*pre-order*, *in-order*, *out-order*, *post-order*) that implement these actions in different orders.

Algorithm 4: Pre-order traversal (NLR)

```
void preorder(N):
    begin
        if N != NULL then
            visit(N);
            preorder(N.left);
            preorder(N.right);
        end
    end
```

Algorithm 5: In-order traversal (LNR)

```
void inorder(N):
    begin
        if N != NULL then
            inorder(N.left);
            visit(N);
            inorder(N.right);
        end
    end
```

Algorithm 6: Out-order traversal (RNL)

```
void outorder(N):
    begin
        if N != NULL then
            outorder(N.right);
            visit(N);
            outorder(N.left);
        end
    end
```

Algorithm 7: Post-order traversal (LRN)

```
void postorder(N):
    begin
        if N != NULL then
            postorder(N.left);
            postorder(N.right);
            visit(N);
        end
    end
```

In a pre-order traversal, every node is visited before its children. This is an example of a *topological sort* or *topological ordering*. In an in-order traversal, nodes are visited left to right. For a binary search tree, an in-order traversal would process nodes in sorted order. Out-order is the opposite of in-order. Nodes are visited right to left, and a binary search tree would be processed in reverse sorted order. In a post-order traversal, every node is visited after its children. It is often used to delete an entire tree, node by node.

DFS can also be applied to graphs. Instead of having a left and right path, a node in a graph may have many paths, each of which needs to be explored completely before moving on to the next. However, depth-first searching a graph with a cycle will result in looping through the nodes in that cycle indefinitely. This can be dealt with by ignoring previously visited nodes and/or by stopping the search at a certain depth. *Iterative deepening* is a process that involves depth-first searching to a certain depth and repeating the search with a deeper depth until a given node is found. Recursive and iterative DFS algorithms are given below. They handle cycles by keeping track of visited nodes.

Algorithm 8: DFS (recursive)

```
Data: A graph G and a node u in G
void dfs-recursive(G,u):
    begin
        Visit u;
        foreach node v in G.adjacentNodes(u) do
            if v is not visited then
                dfs-recursive(G,v);
            end
        end
    end
```

Algorithm 9: DFS (iterative)

```
Data: A graph G and a node u in G
void dfs-iterative(G,u):
    begin
        Let S be a stack;
        S.push(u);
        while S is not empty do
            u ⟵ S.pop();
            if u is not visited then
                Visit u;
                foreach node v in G.adjacentNodes(u) do
                    S.push(v);
                end
            end
        end
    end
```

It is important to note that the recursive and iterative implementations of DFS do not search in the same order. For example, if a binary tree is traversed using recursive DFS, nodes at the same depth will be visited left to right. If it is traversed using iterative DFS, nodes at the same depth will be visited right to left because storing them with a stack will invert their order. It seems then that it might be *preferable* to implement DFS recursively.

Table 17: DFS Computational Complexities

Resource	Complexity	Reasoning
Time	$O(\|V\| + \|E\|)$	The search must visit every node and walk every path.
Space	$O(\|V\|)$	The call stack would hold $\|V\|$ frames if the tree was a path. The stack would have to hold $\|V\| - 1$ nodes if the root were connected to every other node.

10.2 Breadth-First Search

Breadth-first search (BFS) is another way of traversing trees and graphs. Like DFS, it is often used as a method of searching trees, but in general it is just a method to visit every node in a tree or graph. Unlike DFS, it visits all neighbors of a node before visiting deeper nodes.

Unlike DFS, BFS does not have a variety of orders like pre-order, in-order, etc. when applied to a tree. Typically, nodes in the same level will be visited left to right and shallow levels will be processed before deep levels, but the latter rule is the only true requirement of a BFS. Like DFS, BFS can be written recursively, but it is much more natural to write it iteratively, so that algorithm will be given first.

Algorithm 10: BFS (iterative)

```
Data: A graph G and a node u in G
void bfs-iterative(G,u):
    begin
        Let Q be a queue;
        Q.enqueue(u);
        while Q is not empty do
            u ⟵ Q.dequeue();
            if u is not visited then
                Visit u;
                foreach node v in G.adjacentNodes(u) do
                    Q.enqueue(v);
                end
            end
        end
    end
```

Note that the only difference between the iterative implementations of DFS and BFS is the choice of data structure. DFS uses a stack to hold the next nodes to visit while BFS uses a queue. This explains why DFS is natural to write recursively, and BFS is not. In `dfs-iterative`, a stack is used to store the next nodes to visit. In `dfs-recursive`, the algorithm stores its recursive subroutines on the *call stack* which is a stack data structure. Instead of storing the next nodes to visit, `dfs-recursive` stores a subroutine that will find and visit the next nodes.

BFS can be written using recursion, but it will essentially require implementing a stack (the call stack) into the algorithm. This is not a very *meaningful* way to implement BFS. In fact, it is more like implementing DFS and running it on a single-branch tree in which each node contains the BFS queue's state at some particular step during a

regular, iterative BFS. Despite its inherent awkwardness, a recursive implementation of BFS is shown below.

Algorithm 11: BFS (recursive)

```
Data: A graph G and a node u in G
void bfs-recursive(G,u):
    begin
        Let Q be a queue;
        Q.enqueue(u);
        bfs-recursive-helper(G,Q);
    end
Data: Q
void bfs-recursive-helper(G,Q):
    begin
        if Q is not empty then
            u ⟵ Q.dequeue();
            Visit u;
            foreach node v in G.adjacentNodes(u) do
                Q.enqueue(v);
            end
            bfs-recursive-helper(G,Q);
        end
    end
```

The space complexity of this implementation is particularly bad. If you needed to visit $|V|$ nodes, you would have to store $|V|$ queues at once. In a binary tree, when a full node is visited, one node is dequeued and two are enqueued, which increments the size of the queue by 1. When the node has one child, the queue does not change size. When the node has no children, the queue's size decrements by 1. In the worst-case (which occurs if the tree is perfect), the queue would increment in size by 1 until the last level. As BFS visits the nodes in the last level, the queue would decrement by 1, starting at a size of $\log |V|$ and ending empty when the final node is explored. At this point, the call stack would start unwinding. The average queue size in this case would be $^1/_2 \cdot \log |V|$, so the space complexity would be $O(|V| \log |V|)$ instead of the $O(|V|)$ space complexity achieved by the iterative implementation.

Table 18: BFS Computational Complexities

Resource	Complexity	Reasoning
Time	$O(\|V\| + \|E\|)$	The search must visit every node and walk every path.
Space	$O(\|V\|)$	The queue would have to hold $\|V\| - 1$ nodes if the root were connected to every other node.

DFS and BFS are Intertwined

DFS and BFS are two sides of the same coin. DFS is more natural to write recursively, and BFS is more natural to write iteratively. DFS uses a stack, and BFS uses a queue. Both searches are $O(|V| + |E|)$, so they will theoretically finish traversing the same tree in the same amount of time. However, in general, DFS will search for nodes that are far away from the source before BFS will. And, in general, BFS will search for nodes that are near the source before DFS. If you have an idea of "how distant" your desired item likely is from your source, it might be worth choosing one over the other to improve your search performance.

The time complexity of DFS and BFS on trees can also be expressed as $O(b^d)$ where b is the branching factor of the tree and d is the depth of the tree. This suggests that a search is faster if the tree is narrower or shorter. To put it another way, the search is faster the closer the desired node is to the top and the left of the tree.

DFS and BFS are used to solve a variety of fundamental computer science problems. Either one can be used to find the number of connected components in a graph. The idea is to iterate over every node in a graph and, if it has not yet been visited, to perform a DFS or BFS. The number of searches made equals the number of connected components. DFS and BFS can also be used to test if a graph is bipartite. This is done by checking if the graph can be 2-colored. DFS is used to solve mazes. BFS is used to solve shortest path problems.

It is interesting to think of DFS and BFS as two different kinds of personalities. DFS is like a person who is overly confident and always pressing forward, even if they are going in the wrong direction. They only take a step back if they are forced to. BFS is like a person who is overly cautious and always investigating every option before moving forward. They only move on if there is nothing else to learn around them. Both approaches are useful, but they are useful in different scenarios.

10.3 Bidirectional Search

A bidirectional search involves performing two searches on the same graph simultaneously. They can be DFS or BFS, but they are typically BFS. This algorithm is often used to check if two nodes are connected or to solve the shortest path problem. If two breadth-first searches start at two different roots, they will search outward until one reaches a node that the other visited already. If this happens, the two roots are connected, and the shortest path goes through that intersection. The shortest path can be traced from that intersection by moving up through the intersection node's ancestry. In one search, the intersection's oldest ancestor is s. In the other search, it is t. If you wish to know not only that s and t are connected but also the details of their shortest path, you must keep track of the parent of every node you visit, perhaps with a map.

Why is this more useful approach than a regular BFS? Let s and t be two nodes that are d edges away from each other. If you BFS starting from s, you would have to search to a depth of d to find t, which would result in an $O(b^d)$ search time. However, if you BFS starting from both s and t, each process would have to search to a depth of $d/2$ to

find the intersection, which would result in an $O(2b^{d/2}) = O(b^{d/2})$ search time, which is a major improvement.

Algorithm 12: Bidirectional Search

```
Data: A graph G and node s and t in G
void bidirectional(G,s,t):
    begin
        Let sParentMap and tParentMap be maps;
        Let sQ and tQ be queues;
        Let sParent and tParent be null nodes;
        Let sCurr and tCurr be null nodes;
        sQ.enqueue(s);
        tQ.enqueue(t);
        while sQ is not empty and tQ is not empty do
            sCurr ⟵ sQ.dequeue();
            tCurr ⟵ tQ.dequeue();
            sParentMap.put(sCurr, sParent);
            tParentMap.put(tCurr, tParent);
            if sCurr is visited then
                printShortestPath(sCurr,sParentMap,tParentMap); return;
            end
            else if tCurr is visited then
                printShortestPath(tCurr,sParentMap,tParentMap); return;
            end
            Visit sCurr;
            Visit tCurr;
            for node sAdj in G.adjacentNodes(sCurr) do
                sQ.enqueue(sAdj);
            end
            for node tAdj in G.adjacentNodes(tCurr) do
                tQ.enqueue(tAdj);
            end
            sParent ⟵ sCurr;
            tParent ⟵ tCurr;
        end
    end
```

Consider using a bidirectional BFS to solve the search problem instead of the shortest path problem. A breadth-first search will find nearby nodes quickly. Therefore, if you had multiple guesses of where your search item might be located, it would make sense to perform breadth-first searches in all of those areas.

10.4 Dijkstra's Algorithm

Blah

10.5 Binary Search

Blah

10.6 Rabin-Karp Algorithm

Blah

11 SORTING

Blah

11.1 Selection Sort

Blah

11.2 Insertion Sort

Blah

11.3 Merge Sort

Blah

11.4 Quick Sort

Blah

11.5 Radix Sort

Blah

11.6 Topological Sort

Blah

12 MISCELLANEOUS

Blah

12.1 Cache Replacement Algorithms

Blah

12.2 Permutations

Blah

12.3 Combinations

Blah

12.4 Bit Manipulation

Blah

Implementation, Featuring an Exploration of Java

Blah

13 HOW DOES JAVA WORK?

14 JAVA STANDARD LIBRARY

14.1 java.lang

14.2 java.util

14.3 java.io

14.4 java.net

15 JAVA TECHNIQUES

Blah

15.1 Static Initialization Blocks

Blah

15.2 Lambda Expressions

Blah

16 JAVA FRAMEWORKS

Additional Technologies

"Stupid Computer Shit"

17 UNIT TESTING

Blah

17.1 JUnit

Blah

18 VERSION CONTROL

Blah

18.1 Git

How do you download stuff from GitHub? There are a few methods that might be available, depending on the software.

- git clone the directory
- wget raw files
- Grab it from a package repo with something like pacman
- Get it from the AUR with something like yay or yaourt
- Download a tarball, unzip it, extract it, and build the source files into an executable

19 BUILD AUTOMATION

19.1 Make

GNU Make compiles source files into executables.

19.2 Maven

Blah

20 UNIX

20.1 History of Unix and GNU/Linux

20.2 A Tour of the Unix File System

20.3 Common Commands and Tasks

- chmod +x
- .bash_profile is executed at login for the current user
- .bashrc is executed every time a shell is opened for the current user
- INI files
- Run command (rc) files.
- Shell scripts. Shebangs.
- Config files (plain text)
- Handling swap files in Vim. You accidentally deleted a terminal where you were editing a file, and now you have a file with a previous save and the autosaved file. Which one do you want to look at? You probably want to recover (R) the swap file, save its changes to the main file (:w), reload the file content (:e), and when prompted about the existence of a swap file, delete the swap file (D). If the delete option is not available, that means that the file is being edited elsewhere. Go close those windows.
- Daemon - background process
- File Descriptors

20.4 Making a Personalized Linux Installation

- Install Arch
- Download Desktop Environment
- Download Window Manager
- Download Login Manager (or use startx)
- Learn pacman
- Download from the AUR
- urxvt terminal has its perl extensions and configuration in ~/.Xresources (run xrdb ~/.Xresources to have the window system grab the changes without a reboot)
- urxvt needs monospaced font and fonts that support Unicode
- feh
- compton
- polybar
- vim
- Change desktop environment on startup by adding exec *ds_name* to the bottom of ~/.xinitrc

21 VIRTUALIZATION

Blah

22 CONTAINERIZATION

Blah

22.1 Docker

Blah

23 MARKUP AND STYLE LANGUAGES

23.1 TeX

Blah

23.2 HMTL

Blah

23.3 CSS

Blah

24 DATA FORMATS

Blah

24.1 XML

Blah

24.2 JSON

Blah

25 QUERY LANGUAGES

25.1 SQL

Blah

Software Development

26 SOFTWARE ENGINEERING PROCESSES AND PRINCIPLES

- Agile
- Test Driven Development
- "Software development process"
- Single Responsibility Principle
- Open Closed Principle

27 DESIGN PATTERNS

- Singleton
- Abstract Factory

Appendices

A SYSTEM DESIGN

Blah

A.1 Server Technologies

DOMAIN NAME SYSTEM (DNS) SERVER Translates a domain name to an IP address of a server containing the information on the requested website. Could use round-robin, load balancing, or geography to choose a server associated with a certain domain name. An OS or browser can cache DNS results until the result's *time to live* (TTL) expires.

If a local DNS server does not know the IP address of some domain name, it can ask a nearby, larger DNS server if it knows. The biggest DNS servers are called *root servers*, and they are distributed across the world's continents. They return lists of servers that would recognize your requested domain name. These are *authoritative name servers* for the appropriate *top-level domain* (.com, .org, .edu, .ca, .uk, etc.).

There are many kinds of results that can be returned by a DNS server:

NS RECORD Name server. Specifies the names of the DNS servers that can translate a given domain.

MX RECORD Mail exchange. Specifies the mail servers that will accept a message.

A RECORD Address. Points a name to an IP address.

CNAME Canonical. Points a name to another name (google.com to www.google.com) or to an A record.

LOAD BALANCER Decides which servers to send requests to based on some criteria (random, round-robin, load, session, Layer 4, Layer 7). Can be implemented in software or hardware. Can generate cookies to send to the user. The user can then send those cookies back to return to the server they were using. Multiple load balancers are needed for horizontal scaled systems.

REVERSE PROXY A web server that acts as an intermediary between clients and backend servers. Forwards requests to the backend and returns their responses. This hides the backend server IPs from the client and allows for a centralized source of information. However, it is a single point of failure, so load balancers are a better choice for horizontally scaled systems.

APPLICATION LAYER It may be useful to have a layer of application servers separate from your web servers. This allows you to scale the layers independently. A web server serves content to clients using the HTTP. An application server hosts the logic of the application, which could generate an HTTP file to send to a web server. Web servers are often used as reverse proxies for application servers.

A.2 Persistent Storage Technologies

CACHES Caching involves putting data that is referenced often on a separate, small memory component. For example, when you stay on the same domain, your OS can cache an IP address so it doesn't have to look up the same IP address every time.

FILE-BASED CACHING The data that you want is saved in a local file. For example, you could cache an HTML file instead of dynamically creating a website with data from a database. Not recommended for scalable solutions

IN-MEMORY CACHING Copy the most popular pieces of data from an application's database and put it in RAM for faster access times. If you choose to cache a database query as a key and the result as a value, it is hard to determine when to delete this pair when the data becomes stale. Alternatively, you could cache objects. When an object is instantiated, make the necessary database requests to initialize values, and store the object in memory. If a piece of data changes, delete all objects that use that piece of data from cache. This allows for *asynchronous processing* (the application only touches the database when creating objects). Popular systems include Memcached and Redis.

CACHE UPDATE STRATEGIES

CACHE-ASIDE The cache does not interact with storage directly. The application looks for data in the cache. Upon a cache miss, it finds its data in storage, adds it to the cache, and returns the data. Only requested data is cached. Data in the cache can become stale if it is updated in storage but not in cache.

WRITE-THROUGH The application uses the cache as its primary data store, and the cache reads and writes to the database. Application adds data to cache, cache writes to data store and returns value to the application. Writes are synchronous and slow, but data is consistent. Reads of cached data are fast. However, most data written to the cache will not be read.

WRITE-BACK The application adds data to the cache, and the data is asynchronously written to the database. Data loss could occur if the cache fails before new data is written to the database.

RAID *Redundant Array of Independent Disks* is a technique that uses many physical disk drives to improve the redundancy or performance of a system.

RAID0 Writes a portion of a file on one drive and the other portion on another drive concurrently. This doubles write speed but has no redundancy.

RAID1 Writes the whole file on both drives. No write speed improvements, but improves redundancy.

RAID10 A combination of RAID0 and RAID1. If you have 4 drives, a file is striped between 2 drives and the same striped data is written concurrently to the other 2 drives.

RAID5 Given N drives, you stripe data across N-1 drives and stores a full copy of the file on 1 drive.

RAID6 Given N drives, you stripe data across N-2 drives and stores a full copy of the file on 2 drives.

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS) A *relational database* is a collection of items organized in tables. A *database transaction* is a change in a database. All transactions are ACID: Atomic (all or nothing), Consistent (moves the database from one valid state to another), Isolated (concurrent transactions produce the same results as serial transactions), and Durable (changes do not revert). Various techniques for scaling databases are described below.

REPLICATION

MASTER-SLAVE REPLICATION A master serves reads and writes, replicating writes to the slaves, which can only serve reads. If a master goes offline, the system is read-only until a slave is promoted.

MASTER-MASTER REPLICATION There are multiple masters that can serve both reads and writes and coordinate with each other on writes. A master can fail and the system can still be fully-functional. However, writes need to be load balanced. Master-master systems are usually either eventually consistent (not ACID) or have high-latency, synchronized writes.

REPLICATION DISADVANTAGES If a master dies before it can replicate a write, data loss occurs. Lots of write replication to slaves means that slaves cannot serve reads as effectively. More slaves means more replication lag on writes.

FEDERATION Splits up databases by function instead of using a monolithic database. Reads and writes go to the appropriate database, resulting in less replication lag. Smaller databases can fit a greater percentage of results in memory, which allows for more cache hits. Parallel writing is possible between databases. Not effective for large tables.

SHARDING Data is distributed across different databases (shards) such that each database can only manage a subset of the data (like submitting tests to piles labeled A-M and N-Z). Less traffic, less replication, more cache hits, parallel writes between shards. If one shard goes down, the others can continue to work (however, replication is still necessary to prevent data loss). Load balancing between shards is important. Sharing data between shards could be complicated.

DENORMALIZATION Improves read performance at the expense of write performance. Redundant copies of data are written in multiple tables to avoid expensive joins.

SQL TUNING Benchmark your database system and optimize it by restructuring tables and using appropriate variables.

NOSQL A NoSQL database stores and retrieves data in ways other than tabular relations. Its transactions are BASE: Basically Available (the system guarantees availability), Soft state (the state of the system may change over time, even without input), and eventually consistent (will become consistent over a period of time, if no further input is received). NoSQL prioritizes availability over consistency. Some configurations are described below.

KEY-VALUE STORE Stores data using keys and values. O(1) reads and writes. Used for simple data model or for rapidly changing data, such as a cache. Complex operations are done in the application layer.

DOCUMENT STORE All information about an object is stored in a document (XML, JSON, binary, etc.). A document store database provides APIs or a query language to query the documents themselves.

WIDE COLUMN STORE The basic unit of data is a *column* (name/value pair). Columns can be grouped in column families. Super column families can further group column families. Useful for very large data sets.

GRAPH DATABASE Each node is a record and each edge is a relationship between records. Good for many-to-many relationships. Not widely used and relatively new.

A.3 Network Techniques

HORIZONTAL SCALING Distributes your data over many servers. Alleviates the load on a single server, but now requests have to be distributed across these servers. Introduces complexity: load balancers are required, and servers should now be stateless.

ASYNCHRONISM Asynchronous tasks are done to prevent the user from waiting for their results. One example is anticipating user requests and pre-computing their results.

Another example is having a worker handle a complicated user job in the background and allowing the user to interact with the application in the meantime. The worker will then signal when the job is complete. A job could also "appear" complete to the user, but require a few additional seconds to actually complete.

FIREWALLING A network security system typically used to create a controlled barrier between a trusted internal network and an untrusted external network like the Internet. For example, if you want your server to listen for HTTP and HTTPS requests, you could restrict incoming traffic to only ports 80 and 443. This prevents clients from having, for example, full read and write access to your databases.

CONSISTENCY PATTERNS When many servers hold copies of the same data, we must find an acceptable method of updating them.

> **WEAK CONSISTENCY** After a write, reads may or may not see it. A best effort approach is taken. Works when the application *must* continue running and data can be lost during outages (VoIP, video chat, multiplayer games). Good for any sort of "live" service.
>
> **EVENTUAL CONSISTENCY** After a write, read will eventually see it. Works when the application *must* continue running, but data cannot be lost, even during outages (email, blogs, Reddit). Good when writes are important, but reading stale data for a short period of time is acceptable.
>
> **STRONG CONSISTENCY** After a write, reads will see it. Works when everyone needs to see the most up-to-date information at all times, even if it slows the whole system down (file systems, databases). Good when stale data is unacceptable.

SHARED SESSION STATE If users can access a website from many different servers, how do you keep track of their session data? If a user logs in on one server, how can the network know they are logged in when they move to another server? Store all session data on a single server. Use RAID for redundancy.

MICROSERVICES An application can be structured as a collection of microservices that have their own well-defined independent functions. These microservices can be combined in a modular fashion to create the full application. Each microservice could have its own network architecture. This allows for modular scaling of an application. Software like Apache Zookeeper is used to keep track of microservices and how they interact.

CONTENT DELIVERY NETWORK (CDN) A CDN is a globally distributed network of proxy servers that serves content to users from nearby nodes. A *push CDN* only updates some piece of data when the developer pushes data to it. It's faster, but requires more storage on the CDN. A *pull CDN* get content from the developer's server whenever a client requests it. It's slower, but requires less space on the CDN.