The Science of Computation

Alejandro Piad Morffis, Ph.D.

Contents

Prologue: The Quest to Answer all Questions

Humanity's ingenuity extends far back into the foggy ages of the agricultural revolution, and probably even farther back. Since the dawn of Homo sapiens, our species has always been interested in predicting the future. Tracking the seasons, planning crops, estimating an enemy's forces, and building cities, all of these tasks require significantly complex computations that often need to be reasonably accurate to be helpful.

To tackle these problems, humanity invented mathematics, our greatest collective intellectual achievement. Nowadays, it is easy to think of math as this coherent body of knowledge that can be built from the ground up –much like you learn it in school– where every new concept, theorem, or technique, is beautifully derived from previous concepts and theorems. The whole edifice of math seems like a divine gift, a perfect tool for solving the most complex problems.

But this wasn't always the case. Mathematics as we know it today is a very recent invention. Before the 20th century, we had several chunks of math that weren't really all that compatible with each other. So much so, that finding bridges between two different mathematical theories (or fields) has been one of the most fruitful, and challenging work of some of the most eminent mathematicians of all time.

The first big block of mathematical knowledge we came up with is Euclidean geometry, developed by ancient Greeks (and most likely many other ancient civilizations as well). Euclid, a Greek mathematician, was the first to sit down and systematize the existing knowledge in a book, full of axioms, definitions, theorems, and proofs. From Euclid we inherited the idea that mathematical proofs are sequences of claims based on previous claims, that start with some axioms and definitions.

What's missing:

- Logic from Aristotle
- Algebra from the Arabs
- Arithmetics from India

- Decartes and the analytical geometry
- Calculus by Leibniz and Newton
- The state at the early 20th century

A Universal Language for Reasoning

The modern history of Computer Science can be said to begin with Gottfried Leibniz, one of the most famous mathematicians of all time. Leibniz worked on so many fronts that it is hard to overestimate his impact. He made major contributions to math, philosophy, laws, history, ethics, and politics.

He is probably best known for co-inventing, or co-discovering, calculus along with Isaac Newton. Most of our modern calculus notation, including the integral and differential symbols, is heavily inspired by Leibniz's original notation. Leibniz is widely regarded as the last universal genius, but in this chapter, I want to focus on his larger dream about what mathematics could be.

Leibniz was amazed by how much notation could simplify a complex mathematical problem. Take a look at the following example:

"Bob and Alice are brothers. Bob is 5 years older than Alice. Two years ago, his age was twice as hers. How old are they?"

Before Algebra was invented, the only way to work through this problem was to think hard, or maybe try a few lucky guesses. But with algebraic notation, you just write some equations like the following and then apply some straightforward rules to obtain an answer.

Any high-school math student can solve this kind of problem today, and most don't really have the slightest idea of what they're doing. They just apply the rules, and voila, the answer comes out. The point is not about this problem in particular but about the fact that using the right notation –algebraic notation, in this case– and applying a set of prescribed rules –an algorithm– you can just plug in some data, and the math seems to work by itself, pretty much guaranteeing a correct answer.

In cases like this, *you* are the computer, following a set of instructions devised by some smart "programmers" that require no creativity or intelligence, just to painstakingly apply every step correctly.

Leibniz saw this and thought: "*What if we can devise a language, like algebra or calculus, but instead of manipulating known and unknown magnitudes, it takes known and unknown truth statements in general?*"

In his dream, you would write equations relating known truths with some statements you don't know, and by the pure syntactic manipulation of symbols, you could arrive at the truth of those statements.

Leibniz called it *Characteristica Universalis.* He imagined it to be a universal language for expressing human knowledge, independently of any particular language or cultural constraint, and applicable to all areas of human thought. If this language existed, it would be just a matter of building a large physical device –like a giant windmill, he might have imagined– to be able to cram into it all of the current human knowledge, let it run, and it would output new theorems about math, philosophy, laws, ethics, etc. In short, Leibniz was asking for an *algebra of thoughts*, which we today call **logic**.

He wasn't the first to consider the possibility of automating reasoning, though. Having a language that can produce true statements reliably is a major trend in Western philosophy, starting with Aristotle's syllogisms and continuing through the works of Descartes, Newton, Kant, and basically every single rationalist that has ever lived. Around four centuries earlier, philosopher Ramon Llull had a similar but narrower idea which he dubbed the *Ars Magna*, a logical system devised to prove statements about, among other things, God and the Creation.

However, Leibniz is the first to go as far as to consider that all human thought could be systematized in a mathematical language and, even further, to dare dream about building a machine that could apply a set of rules and derive new knowledge automatically. For this reason, he is widely regarded as the first computer scientist in an age where Computer Science wasn't even an idea.

Leibniz's dream echoes some of the most fundamental ideas in modern Computer Science. He imagined, for example, that prime numbers could play a major part in formalizing thought, an idea that is eerily prescient of Gödel's numbering. But what I find most resonant with modern Computer Science is the *equivalence between a formal language and a computational device*, which ultimately becomes the central notion of computability theory, a topic we will explore in future chapters.

Unfortunately, most of Leibniz's work in this regard remained unpublished until the beginning of the 20th century, when most of the developments in logic needed to fulfill his dream were well on their way, pioneered by logicians like Gottlob Frege, Georg Cantor, Bertrand Russell.

We will look at that part of the story next.

The Rules of Thought

Frege, Boole, and Cantor.

The Final Frontier of Math

Hilbert's program to solve all mathematical problems.

Crisis in Infinite Math

The crisis of math that started with Russell and then Gödel We end the prologue here, just before Turing.

Introduction

This book takes you through a journey across the most important fields and themes in Computer Science. We will meet many of its most important figures –from Alan Turing, Edsger Djikstra, and Donald Knuth, to Geoffrey Hinton and Yan LeCunn– and see how they came up with some of the most brilliant ideas in the science of computation.

We will dive into some the most fascinating discoveries in Computer Science, from the neural networks that power large language models, to the packet switching protocols that make Internet possible. We will learn how computers work, in and out, why are they so powerfull, but also what are their intrinsic limitations. In the end, you will gain a big-picture understanding of the whole field of Computer Science.

This book contains no complicated math or code. There are absolutely no prerequisites to read it, although it is perhaps better suited for highschool-level students or older, because some of the concepts in Computer Science are somewhat sophisticated and rely on intuitions that students form around that age. It is also a perfect companion for a Computer Science major, as it complements the more theoretically- or technically-oriented bibliography you will encounter in college textbooks.

A Map to the Science of Computation

Computer Science is a relatively new field that emerged in the early 20th century from a massive effort to formalize Mathematics. However, it wasn't until the mid-60s and early 70s that it gained recognition as a distinct and unique scientific field. This was achieved by establishing the theoretical grounds of computability and complexity theory, which proved there were many interesting, novel, and challenging scientific questions involving computation.

Computer Science is an extensive field that encompasses a wide range of knowledge and skills. It goes from the most abstract concepts related to the nature of computable problems to the most practical considerations for developing useful software. To solve problems in this field, one must navigate through various

levels of abstraction, making it challenging to provide a concise summary that the average reader can easily understand.

Computer Science encompasses a vast range of subjects, incorporating multiple areas of math and engineering. It also introduces many original concepts and ideas. Thus it is impossible to divide this field objectively into subfields or separate areas with precise borders. I will attempt to categorize it into somewhat cohesive disciplines, but keep in mind there is considerable interconnectedness between them.

This book is divided into five parts, each subsequently divided into several chapters, encompassing all of Computer Science. The following sections will give you a taste of what you can find in each section and chapter, with links (in **bold** font) to each relevant section so you can quickly jump to your preferred content. Do keep in mind the book is best read top-to-bottom, as some of the later ideas rely on earlier content. However, if you want to learn about a particular subject, feel free to hop and shop around. Whenever some earlier idea is important to understand what your reading, you will find useful links.

Foundations of Computation

We'll start our journey by looking at the foundations of Computer Science. The fundamental question of Computer Science is something akin to *"What kinds of problems can be solved with algorithms."* It was first asked formally by David Hilbert in 1901 and first answered by Alan Turing in 1936. This central question of **Computability Theory** establishes the foundational theory for what can be done with any computer.

The core concept in computability is the **Turing machine**, an abstract computer model that we define to study the properties of computation and computable problems irrespective of any concrete machine our current technology supports. Turing machines let us answer which kinds of problems are, in principle, undecidable —which means that no algorithm can ever be devised to solve them completely. Surprisingly, there are many of those, not all esoteric; there are very practical problems in Computer Science that we know are mathematically impossible to solve. Turing machines also give a precise definition of **algorithm**.

Once we lay out the limits of computation, we can ask which computable problems are easier or harder. **Complexity theory** deals with the complexity of different problems. It asks how efficiently, most commonly in terms of computing time and memory, we can solve any problem. For some problems, we can even prove that we have found the most efficient algorithm anyone could ever develop. The most important problem in complexity theory, and probably in the whole of Computer Science, is the famous question of whether $\mathbf{P} = \mathbf{NP}$, which is ultimately a question about the nature of really hard problems.

The final ingredient in the foundational theory of Computer Science is the surprising relationship between languages and machines. **Formal language**

theory allows us to formalize the notion of a language, whether it is a human language like English or Spanish, a programming language like Python or C#, and any technical, abstract, or mathematical notations used in many fields.

The basic concept in formal language theory is a formal grammar, a mathematical definition of what can be said with a specific language. In contrast, automata theory deals with abstract machines —like Turing machines— of different levels of complexity that can recognize specific families of formal languages. The study of the relationship between language generation and recognition led to the invention of **compilers**, which are computer programs that understand specific programming languages and translate them to machine code that can be run on concrete hardware.

But our journey through the foundations doesn't end here. Computer Science emerged from Mathematics, and therefore, it adopted the mathematical practice of establishing axioms and demonstrating theorems. Besides, when tackling real-world computational challenges, we frequently apply mathematical techniques used in engineering and other fields of science. Consequently, a substantial amount of mathematical knowledge underpins Computer Science, albeit with a computational perspective. Thus, to complete this first part, we will look at the foundations of **Computational mathematics**.

Logic forms the bedrock of Computer Science as it gave birth to its fundamental principles. By enabling us to formalize reasoning algorithmically, Logic allows us to create programs that can manipulate logical formulas and prove theorems and also lets us formally prove that an algorithm works as intended. **Discrete math** is a natural extension of Logic, encompassing a vast mathematical field that studies discrete objects such as collections of natural numbers. It formalizes numerous basic computational concepts like **graphs** and **trees**. Additionally, Discrete Math lies at the core of modern cryptography, which relies heavily on number theory.

Algorithms and Data Structures

Beyond the theoretical foundations, solving a concrete Computer Science problem will often require using one or more algorithms. Hence, studying which specific algorithm can solve which concrete problem is a big part of Computer Science. In the second part of the book you'll learn about some of the most basic and commonly used algorithms and data structures in Computer Science. We will start by looking at *arrays* and *strings*.

Strings are sequences of symbols —called *characters*— most commonly used to represent text. **String matching**, i.e., the problem of finding if a given pattern appears in a text, is one the most studied problems in CS with applications everywhere from compiler design to DNA analysis to chatbots to search engines.

An **array** is a collection of items of an arbitrary type —e.g., numbers, words, images, or custom user data records. The most basic operations in arrays

are **searching and sorting**, which give rise to some of the most clever algorithms ever designed. On top of arrays, we find many **data structures** that focus on efficiently performing some particular operations on specific types of objects. These include *lists*, *queues*, *stacks*, *heaps*, and the two fundamental ways to represent data in Computer Science: *trees* and *graphs*.

A **tree** is a hierarchical collection of items in which every item has a *parent* and potentially many *children*. It is used anywhere we need to represent intrinsically hierarchical data, such as organizations of people. However, trees also appear in many problems where the hierarchy is not obvious, such as evaluating mathematical expressions or searching large mutable collections of objects.

Graphs are the most general data structure to represent abstract information computationally. A graph is a collection of nodes connected by edges. It can represent anything network-like, from geospatial data to social networks to abstract things like the dependencies between subtasks in a given problem. Graph algorithms are so vast that you can find whole subfields dedicated to just a subset of them. Still, the most basic one is **path-finding**, i.e., finding a path —often optimal in some sense— between one specific source node and one or more destinations.

However, just memorizing a huge toolbox of algorithms is not enough. Sometimes you'll find problems for which no one has already designed an algorithm. In these cases, you'll need **algorithm design strategies**, which are higher-level ideas and patterns useful for creating new algorithms.

The most powerful such strategy is **recursion**, so much so that *"you can do anything with recursion"* is a popular meme in Computer Science. Recursion is all about solving a problem by decomposing it into subproblems that are easier to solve. When taken to its extreme, this idea allows us to formalize the whole theory of computability, which means that any algorithm in Computer Science is, at its core, recursive. There are many recursive patterns; some of the most commonly used are *tail recursion*, *divide and conquer*, and *backtracking*.

Other powerful design strategies underpin many of the most successful algorithms in Computer Science. **Greedy algorithms**, for example, are the ones that make the best short-term choice. While this is often suboptimal, there are surprisingly many cases where a carefully chosen local optimum leads to a global optimum. **Dynamic programming** is another such strategy. Under the right circumstances, it allows transforming a slow recursive algorithm into a very fast non-recursive version while retaining all the correctness guarantees.

Once we finish this part, you will have an intuitive understanding of the most important algorithms and data structures used in all computer applications, from low-level stuff like sending information through the Internet, to higher-level problems like text editors, browsers, and even videogames.

Computational Systems

Mastering algorithms is not enough, though. Ultimately, we need a physical device that can run those algorithms and give us a result at a reasonable cost. Real computers are the physical embodiment of Turing machines, but the devil is in the details. Building a real computer is far more involved than just wiring some cables. Even assuming, as we do in CS, that electronic components always work as expected, the design of resilient computational systems is a whole discipline.

In the third part of the book we will explore how computers, and computational systems in general, are built from the ground up. We will being at the lowest level, understanding how simple electronic components give rise to the marvel that is the device in which you are reading this book (because you're reading it digitally, right?) From there, we'll move up and up until we reach the whole planetary, interconnected network of computers that is the Internet.

At the lowest level, we have transistors, microscopic circuitry that performs a very simple operation: an electronic current breaker. We can design logical circuits —combinations of electronic components that perform whatever logical operation we desire— with just a few transistors. We can design circuits to add, multiply, or store values to build a Central Processing Unit (CPU) connected to a Random Access Memory (RAM) bank. This is the basic computer, which can be programmed with very simple languages called *Assembly Languages*. It is at this level that the frontier between hardware and software melds.

Modern computers have increasingly more complex components, including external memory drives and peripherals like displays and keyboards. Keeping all of those components in harmony is the job of the Operating System. OSes like Windows and Linux do much more, though. They provide an abstraction layer over common operations that many programs need, including interfacing with hardware peripherals —e.g., playing sounds or capturing video— or writing and reading from files. Two key abstractions are *processes*, which allow each program to behave as if it were the only one running, and *threads*, which enable concurrency even if you have a single physical CPU.

But if one computer is incredible, the real magic begins once you have two or more working together. Enter the world of computer networking. The first problem we must solve is simply getting two computers on both sides of the planet, talking reliably to each other over a noisy, error-prone communication channel. The TCP/IP protocol is a collection of algorithms that guarantee communication even when intermediary steps fail constantly. It's the software backbone of the Internet, possibly the single most important technological advance after the first industrial revolution.

Distributed systems allow multiple computers to communicate and collaborate, creating larger organizational levels of computational systems. The simplest setup is the client-server architecture, where an application is split between the user and the service provider. As these systems scale up, they require

coordination among several computers to maintain reliability even when some individual components fail. To achieve this, consensus algorithms are used for data distribution, monitoring system health, and error correction tasks.

Some of today's largest software platforms, like the Google Search Engine and Amazon Web Services, rely on distributed systems as their foundation. These infrastructures are often called cloud computing —vast networks of computers working together as one unified computing platform.

But we won't stop here. A crucial component in computational systems is security, especially when valuable information is stored on computers connected to the internet. Ensuring reliable and secure communication between computers is a major challenge in computer science, addressed with the help of cryptography. Thus, our final stop in the computational systems story will take us down the rabbit hole of World War II secrets.

Cryptography has its roots in breaking machine codes during World War II, such as the Enigma code deciphered by Alan Turing. Modern cryptography relies on *symmetric* and *asymmetric* or public-key methods, which enable two actors to establish a secure channel over an insecure communication medium without ever meeting each other. This principle forms the backbone of most online interactions, from reading your email to using your bank account to chatting with another person so that no one can access your data, often not even the service provider.

Software Engineering

Building software that works is extremely hard. It's not enough to master all the algorithms, data structures, theorems, and protocols. Software products have unique qualities among all other engineering products, especially regarding flexibility and adaptability requirements. Any software that lives long enough will have to be changed in ways that weren't predictable when the software was first conceived. This poses unique challenges to the engineering process of making software, the domain of software engineering. In the fourth part of the book, we will look at all the intricate components that participate in the process of getting an app ready for you to use.

Software construction starts with a programming language, of which there are plenty of variants. Some programming languages are designed for specific domains, while others are called general purpose and designed for any programming task. A fundamental distinction is between *imperative* and *declarative* languages. The most influential paradigms are *functional* (often declarative) and *object-oriented* (most commonly imperative) programming models.

Programming languages are tools, so their effectiveness in solving a concrete problem relies heavily on applying best practices. Software engineering also deals with finding principles and practices to use better the available tools, including technical tools and human resources. The most important set of

software engineering principles is the SOLID principles. Other principles, such as DRY and YAGNI, emphasize specific, pragmatic mindsets about software development. Additionally, we have dozens of design patterns: concrete reusable solutions to common software development problems.

Beyond the actual code, the most crucial resource most applications must manage is data. As the necessity to organize, query, and process larger and larger amounts of data increases, we turn from classic in-memory data structures to more advanced long-term storage solutions: databases. The *relational database* is the most pervasive model for representing and storing application data, especially structured data. However, several alternative non-relational paradigms are gaining relevance as less structured domains (such as human language and images) become increasingly important.

In the data-centric software world, information systems are some of the most complex applications we can find. They provide access to vast amounts of information, which can be from a concrete domain —such as medical or commercial information— or general purpose like search engines. These systems combine complex algorithms and data structures for efficient search with massively distributed hardware architectures to serve millions of users in real-time.

As we've seen, software ultimately runs on someone's computer, and different computational systems pose different challenges. From a software engineering perspective, we can understand those challenges better if we think in terms of *platforms*. The three major software platforms are the *desktop* (apps installed in your computer), the *mobile* (apps installed in your smartphone), and the *web* (apps running on your browser).

Developing desktop software is hard because different users will have different hardware, operating systems, and other considerations, making it hard to build robust and portable software. Mobile development brings additional challenges because mobile devices vary widely and are often more limited than desktop hardware, so performance is even more crucial. The web is the largest platform, and we've already discussed distributed systems. The web is an interesting platform from the software engineering perspective because it is pervasive and lets us abstract from the user's hardware. Perhaps the most famous software development technologies are those associated with web development: HTML, CSS, and JavaScript.

The final component in software engineering concerns the interaction between the user and the application. Human-computer interaction (HCI) studies the design of effective user interfaces, both physical and digital. Two major concerns are designing an appropriate *user experience*, and ensuring *accessibility* for all users, fundamentally those with special needs such as limited eyesight, hearing, or movement.

But before moving on, we will look at one particularly interesting type of software: videogames. We will see how modern game engines work, from physics simulation to graphic rendering. But, more importantly, we will see what is involved in the

process of creating a AAA videogame, which is one of the most challenging and complex types of software currently made.

Artificial Intelligence

Our final stop in this broad exploration of Computer Science, we will look at the amazing field of Artificial Intelligence. AI is a broad and loosely defined umbrella term encompassing different disciplines and approaches to designing and building computational systems that exhibit some form of intelligence. Examples of machines exhibiting intelligent behavior range from automatically proving new theorems to playing expert-level chess to self-driving cars to modern chatbots. We can cluster the different approaches in AI into three broad areas: *knowledge*, *search*, and *learning.*

Knowledge representation and reasoning studies how to store and manipulate domain knowledge to solve reasoning and inference tasks, such as medical diagnosis. We can draw from Logic and formal languages to represent knowledge in a computationally convenient form. Ontologies are computational representations of the concepts, relations, and inference rules in a concrete domain that can be used to infer new facts or discover inconsistencies automatically via logical reasoning. Knowledge discovery encompasses the tasks for automatically creating these representations, for example, by analyzing large amounts of text and extracting the main entities and relations mentioned. Ontologies are a special case of semantic networks, graph-like representations of knowledge, often with informal or semi-formal semantics.

Search deals with finding solutions to complex problems, such as the best move in a chess game or the optimal distribution of delivery routes. The hardest search problems often appear in *combinatorial optimization*, where the space of possible solutions is exponential and thus unfeasible to explore completely. The most basic general search procedures —Depth-First Search (DFS) and Breadth-First Search (BFS)— are exact, exhaustive, and thus often impractical. Once you introduce some domain knowledge, you can apply heuristic search methods, such as A* and Monte Carlo Tree Search, which avoid searching the entire space of solutions by cleverly choosing which solutions to look at. The ultimate expression of heuristic search is metaheuristics —general-purpose search and optimization algorithms that can be applied nearly universally without requiring too much domain knowledge.

Machine learning enables the design of computer programs that improve automatically with experience and is behind some of the most impressive AI applications, such as self-driving cars and generative art. In ML, we often say a program is "trained" instead of explicitly coded to refer to this notion of *learning on its own.* Ultimately, this involves finding hypotheses that can be efficiently updated with new evidence.

The three major paradigms in this field are *supervised*, *unsupervised*, and *reinforcement* learning. Each case differs in the type of experience and/or feedback

the learning algorithm receives. In supervised learning, we use annotated data where the correct output for each input is known. Unsupervised learning, in contrast, doesn't require a known output; it attempts to extract patterns from the input data solely by looking at its internal structure.

Reinforcement learning involves designing systems that can learn by interaction via trial and error. Instead of a fixed chunk of data, reinforcement learning places a learning system —also called an *agent* in this paradigm— in an environment, simulated or real. The agent perceives the environment, acts upon it, and receives feedback about its progress in whatever task it is being trained on. When the environment is simulated, we can rely on different paradigms, such as *discrete events* or *Monte Carlo simulations*, to build a relatively realistic simulation. Reinforcement learning is crucial in robotics and recent advances in large language models.

All of the above are general approaches that can be applied to various domains, from medical diagnosis to self-driving cars to robots for space exploration. However, two domains of special importance that have seen massive improvements recently are vision and language. The most successful approaches in both fields involve using artificial neural networks, a computational model loosely inspired by the brain that can be trained to perform many perceptual and generative tasks. ANNs draw heavily from algebra, calculus, probability, and statistics, representing some of the most complex computer programs ever created. The field of neural networks is alternatively called deep learning_, mostly for branding purposes.

Computer vision deals with endowing computational systems with the ability to process and understand images and videos for tasks like automatic object segmentation and classification. Classic approaches to computer vision rely heavily on signal processing algorithms stemming from algebra and numerical analysis. Modern approaches often leverage neural networks, most commonly convolutional networks_, loosely inspired by the visual parts of animal brains.

Natural language processing (NLP) enables computational systems to process, understand, and generate human-like language. It encompasses many problems, from low-level linguistic tasks, such as detecting the part-of-speech of words in a sentence or extracting named entities, to higher-level tasks, such as translation, summarization, or maintaining conversations with a human interlocutor.

The most successful approaches in modern NLP leverage transformer networks, a special type of neural network architecture that can represent some forms of contextual information more easily than other architectures. These are the mathematical underpinnings behind technologies like large language models and chatbots. So we will finish this part, and the whole book, with a look at modern LLMs, how they work, and what we can expect from them in the near future.

What's next?

Computer science is a vast field encompassing both theoretical and practical aspects. It goes from the most abstract computational concepts grounded in logic and mathematics to the most pragmatic applications based on solid engineering principles to frontier research like building intelligent systems. Working in CS also involves navigating complex social interactions between developers, users, and society and requires awareness of the many ethical issues that automation and computation pose.

Mastering computer science is a significant challenge, making it one of the most demanding academic pursuits. However, this discipline offers immense fulfillment since computation lies at the heart of our modern world. Software pervades all sectors and industries. Those proficient in computer science can find work across various disciplines and not only by creating commercial software. Across all scientific disciplines, from understanding the frontiers of space to solving climate change, designing new materials, and extending human life, every challenging problem in every scientific field has computational aspects that require collaboration between computer scientists and experts from that domain.

This book *will not* make you a computer scientist overnight. No book can. That requires years of learning and training. My purpose is much more humble. All I want is to light up your curiosity about these topics, and perhaps, motivate you to, one day, become a computer scientist yourself. At the very least, I hope that this book serves you to understand the where fundamental ideas in Computer Science come from, and how they permeate the technology you use every day, and shape the world you live in.

Once you finish this book, you will have an intuitive but solidly grounded understanding of the most important ideas in the vast field of Computer Science. But before we finish, we'll have one final meetup. In the Epilogue, we will talk about what the near and not-so-near future holds for Computer Science, how we expect technology to change our society, and what can be your role in this future.

If any of that sounds intriguing, then read on!

Foundations of Computation

Computability Theory

In 1930, Alan Turing aimed to answer this question once and for all. At that time, computers as we know them today did not exist. However, the term "computer" was already used, referring to individuals, primarily women, who would perform computations.

These human computers, often instructed by physicists or mathematicians, were responsible for solving calculation problems. Once the problem was devised and all the reasoning and creativity were complete, these women would receive a piece of paper containing precise instructions for the computation.

Their sole duty was to follow these instructions meticulously. They would manipulate the input numbers, essentially performing the step-by-step process that modern computers carry out. By adhering to the instructions, they would ultimately arrive at the desired answer, even if they didn't understand the underlying math or physics involved.

Now, Turing is a logician living in the early 20th century, and all logicians back then sought to address a crucial question that dominated mathematical research at the time. This question pondered the existence of limitations in mathematics.

Can purely computational, logical reasoning solve every problem? Are there certain questions that lie beyond the realm of what pure logic can answer? Such limitations were hinted at by logicians like Gödel, Russell, and Hilbert. However, Turing approached this problem from a different perspective: computational processing instead of pure logical demonstration.

The Turing machine

Turing's initial idea was to formalize and abstract the concept of a computer. He began by contemplating the nature of human computers. These human computers used a sheet of paper, often divided into cells, on which they wrote various symbols, including numbers. Turing realized the two-dimensional aspect of the paper is not essential. It could be simplified into a one-dimensional tape of cells on which symbols could be placed, deleted, or added at any given moment.

He then observed that these individuals followed instructions, focusing only on one specific instruction at any given time. There was no need to consider multiple instructions concurrently, as they were performed sequentially. Additionally, these instructions were finite and entirely predetermined for a given problem. Regardless of the length of the numbers being multiplied, for example, the instructions for multiplication are always the same.

From these observations, Turing conceived the idea of a computation machine, which he referred to as an "alpha-machine." He imagined this contraption as a state machine with fixed states connected by transitions. These transitions dictated actions based on the observed symbol and current state. For example, if the observed symbol was a 3 and the machine was in state 7, the instruction would specify changing the number to a 4 and transitioning to state 11.

Despite its simplicity, this concept encapsulates the fundamental principle of following instructions and utilizing an external memory, such as a tape, to store intermediate computations. By uniting these concepts, Turing laid the foundation for the conception of a computation machine —an abstract device capable of performing any type of computation.

The alpha-machine, now known as a **Turing machine**, is an abstract representation of a concrete algorithm. For instance, there can be a Turing machine designed explicitly for multiplying two numbers, a Turing machine for computing the derivative of a function, or even a Turing machine for translating from Spanish to English —as hard as that may seem. Each machine is a distinct algorithm akin to a specific program in Python or C++.

The universal computer

But Turing took a further step, introducing the concept of a *universal machine*. This is where the brilliance of his idea shines. Turing deduced that a Turing machine's description can be considered a form of data as well. We can take the states and transitions of a Turing machine and convert them into a long sequence of numbers, which can then be inputted into the tape of another Turing machine.

A universal Turing machine is designed to receive the description of any concrete Turing machine, along with its respective inputs, on its input tape. The universal Turing machine then simulates the precise actions of the specific Turing machine, replicating its behavior with the given input. Turing demonstrated this could be done, thus inventing the modern concept of a general-purpose computer.

And that's how Alan Turing almost single-handedly invented the science of Computation —the theoretical field that explores the limits of computation and its practical implementation, now somewhat confusingly called "Computer Science."

Remarkably, Turing machines, despite their abstract nature, serve as the practical

blueprints for modern computers. In modern computers, a microprocessor contains a fixed set of instructions, while random access memory provides us with a virtually unbounded tape —though not technically infinite. Hence, the concept of a universal Turing machine aligns closely with a modern computer.

However, the story does not conclude here. You see, Turing had two fundamental questions in mind. First, he sought to determine the essence of an algorithm and how we can formalize the concept of computation. That's the Turing Machine we just talked about. But, most importantly, he targeted the most crucial question regarding the limitations of mathematics and computation at the beginning of the 20th century.

The quest to answer all questions

During this time, mathematicians were engaged in a significant undertaking to resolve mathematics altogether. They aimed to address all the remaining critical open questions in the field. A prominent mathematician named David Hilbert compiled a list of 20 questions that he regarded as the most pivotal in mathematics. While many questions on the list pertained to specific areas of mathematics, at least two questions dealt with the fundamental limits of the discipline itself.

The first question asked to **prove the completeness and consistency of mathematics**. This meant establishing that **all truths can be proven** and that mathematics has **no internal contradictions**.

Initially, most mathematicians believed this to be true. However, Kurt Gödel's incompleteness theorem dealt a major blow to this belief. Gödel demonstrated that in any sufficiently strong mathematical system, there are always truths that cannot be proven within that system. To prove them, looking outside the system and introducing new axioms was necessary. This revelation undermined the mathematicians' quest to solve mathematics definitively.

Yet, one fundamental question remained, and it was also widely expected to be possible. This question examined whether, for all provable truths, there existed a completely mechanized procedure—an algorithm—that could find and deliver a proof within a finite amount of time. In simpler terms, it asked if **computers could prove all provable theorems**.

Surprisingly, the answer also turned out negative. There are problems in math and computer science that have clear answers—either true or false—yet cannot be solved algorithmically. These are known as **undecidable problems**: no algorithm can determine the truth or falsehood of these problems for all instances. And that is not a limitation of current technology. *It's a fundamental limitation of the very notion of algorithm.*

Undecidable problems

There are two arguments to understand why computer science must have undecidable problems, using two of the most powerful in logic. The first is a diagonalization argument, very similar to Cantor's original proof that the natural numbers are less than the real numbers.

Essentially, any problem in computer science can be seen as a function that takes some input and produces some output. For example, the problem of adding two numbers can be represented as a function that takes two numbers and produces their sum. The question then becomes whether there are functions that cannot be solved by any theoretical machine or algorithm.

Now, here's the kicker. The number of possible problems, or functions, is uncountable, while the number of Turing machines is countable. We can list them all by enumerating all binary strings and determining which ones correspond to well-formed Turing machines, in the same way in which we can enumerate all Python programs simply by enumerating all possible strings and running the Python interpreter on each one.

So, the number of problems corresponds to the cardinality of real numbers, while the number of programs corresponds to natural numbers. Consequently, there must be infinitely many mathematical problems that cannot be solved by an algorithm.

However, even if this is true, it is conceivable that we may not care about most of these unsolvable problems. They might be unusual or random functions that we do not find significant because for almost every problem we can think of, we can devise an algorithm to solve it. So, while there may be many solvable problems, their importance is subjective. Turing thus aimed to find one specific problem that was both important and unsolvable, and for that, he used the second most powerful tool in the logician's arsenal.

Consider a Turing machine and a specific input. The Turing machine can either run indefinitely in a loop or eventually halt. If the machine halts after a certain number of steps when given that input, we can determine this after a finite amount of time. However, if the machine never halts, we might never be able to tell whether it will halt. It may always be that it has not yet stopped but will do so at some point in the future. Therefore, running a Turing machine that never halts on a given input cannot tell us that it will not halt.

Turing's question, then, is whether it is possible to determine, by examining only the code and input of a Turing machine, whether it will halt or not, *without actually running the machine*. This is known as the **halting problem**.

To establish the undecidability of the halting problem, Turing employs a negated self-reference, a powerful method pioneered by Bertrand Russell with his barber's paradox. The formal proof is somewhat involved, but the basic idea is pretty straightforward.

Following Russell's template, Turing presents a thought experiment assuming the existence of a "magical" Turing machine that can determine if any other Turing machine will halt without executing it. To show this is impossible, Turing pulls a Russell and creates a new Turing machine, using that magical halting machine, that halts when another arbitrary Turing machine doesn't. By running the new machine on itself, Turing builds an explicit contradiction: *the machine must halt if and only if it doesn't halt.* This contradiction proves that the existence of the magical Turing machine for the halting problem is inherently self-contradictory and, therefore, impossible.

What does this mean?

What is the significance of the halting problem? For starters, with this result, Turing not only kickstarted computer science but also established its core limitations on day one: the existence of well-defined problems that cannot be solved by algorithms.

However, although the halting problem may seem abstract, it can be interpreted practically as a fundamental limitation of the kind of software we can build. When writing complex code, we often want a compiler, or linter, to determine if our code is error-free before executing it. For instance, a modern compiler for a statically typed programming language can detect and notify developers of potential type errors before runtime, like using undefined variables or calling unexisting methods.

Ideally, we would want a super smart compiler that could answer questions like: Will this program ever result in a null reference error? Is there a possibility of infinite recursion? Can this program cause an index-out-of-range exception? Unfortunately, these questions all trace back to the fundamental nature of the halting problem; they are, in general, undecidable.

Therefore, the undecidability of the halting problem implies that most of the problems we could expect compilers to solve are fundamentally unsolvable. Consequently, automated program verification —the process wherein programs are checked for intended functionality by other programs— is generally undoable. We must find heuristics and approximate solutions to particular cases because no single algorithm will work on all possible such problems.

This limitation extends to the realm of artificial intelligence as well. These findings tell us that there are fundamental tasks computers will never be able to accomplish. Maybe this means AGI is impossible, but maybe it doesn't. Perhaps humans are also fundamentally limited in this same way. Many believe human brains are just very powerful computers, and if we are ultimately something beyond fancy Turing machines, *we might never be able to know.*

Computational Complexity

There are many problems in computer science for which, even though they seem complicated at first, if we think hard enough, we can come up with clever algorithms that solve them pretty fast.

A very good example is the shortest path problem. This is basically the problem of finding the shortest path between two given points in any type of network. For example, you have a map with millions of places and intersections in your city, and you have to go between one point at one extreme of the map and another point at the other end of the map.

Our cleverest algorithms, on average, have to analyze very few options to get you the shortest path. They are very fast. This is what powers your GPS route planner, for example.

However, other problems can sometimes be very similar to the first type, but no matter how hard we think, we cannot find any clever algorithm that works all the time.

The quintessential example of this is the traveling salesman problem. Consider a similar setting: you have a city and a bunch of places you need to visit in one tour. The question is, what is the best possible tour —the one that takes the least time or travels the least distance?

This problem seems very similar to the shortest path problem, but if you solve it by starting at one point and traveling to the closest next location iteratively, you can end up with a cycle that is several times worse than the best possible cycle.

There are approximate solutions that you can try in very constrained settings. Still, in the general case, no one has ever developed a clever algorithm that solves this problem any faster than checking all possible cycles.

And the thing is, all possible cycles are a huge number.

If you have 20 cities, all possible tours amount to something close to 20 factorial, which is ... (checks numbers) ... well, huge. Not only that, if you have a computer today that can solve the problem for, let's say, 20 cities in one second, then 22 cities will take nearly 8 minutes, 25 cities will take 73 days, and 30 cities will take you... 3 and a half million years!

This is an exponential solution, and they become bad really fast. But for many problems, like the traveling salesman, we have nothing better —that always works.

P and NP problems

The first type of problem, like the shortest path, are called P-problems —technically, *polynomial time complexity*, but don't worry about that. P basically means **problems that are easy to solve.**

Now, for the second type of problem, we don't really know whether they are P-problems, but there is a subset of these that has an intriguing characteristic.

If I ask you to find me a cycle with, say, less than 100 km in total, that will be extremely hard. However, if I tell you, "here is a cycle that takes less than 100 km" you can easily verify it —just add up the length of each road.

These types of problems are called NP problems —technically, *non-deterministic polynomial time complexity*, but again, forget about that. NP basically means **problems that are easy to verify.**

So, the most fundamental question in computer science, P versus NP, ultimately is the following:

P vs NP: *Are all problems that are easy to verify also easy to solve? Or are there problems that are intrinsically harder to solve than to verify?*

Intuitively, for many, it seems it must be the latter. Think about what it would mean to be able to solve easily any problem that is also easy to verify. In a sense, it would be as if recognizing a masterpiece would be the same difficulty as creating the masterpiece in the first place. Being a great critic would be the same as being a great artist.

This seems false, but intuitions in math and computer science are often wrong. We cannot use this kind of analogy to reach any sort of informed conclusion.

Are there really hard problems?

However, theoretical reasons hint at the possibility that P is indeed distinct from NP —that there are some truly, fundamentally difficult problems. The strongest one is the existence of so-called NP-complete problems. Problems, like the traveling salesman, so difficult to solve efficiently that if you could solve one of them, you would solve all the other NP problems at the same time.

When we say a problem is difficult, we mean that it is exponentially easier to verify a solution's correctness than it is to find the solution in the first place. This concept is crucial because, for most problems in computer science, such as sorting, searching for elements in an array, or solving equations, the effort required to solve the problem is roughly the same as the effort needed to verify the solution.

In technical terms, the advantage of verifying over solving is only polynomially easier, not exponentially. However, there are certain problems, like the traveling salesman problem, for which we have been unable to find a fast and efficient algorithm – only exponential ones. Nevertheless, these solutions are very easy to verify.

The heart of the P versus NP debate lies in whether these inherently difficult problems truly exist. Do problems exist that are far more challenging to solve than to verify? To answer this question fully, we would need to find a problem for which a polynomial time algorithm *cannot exist*.

P vs NP remains an unsolved question, and although it has seen a lot of progress, it appears to hint at the need for a new kind of mathematics. Our current mathematical methods lack the power to tackle these challenging meta-questions. However, we do have the next best thing —a wealth of empirical and theoretical evidence suggesting that, indeed, many of these problems may be unsolvable efficiently.

One of the simplest forms of empirical evidence is the vast number of problems for which we lack a polynomial time algorithm to solve them. However, this evidence is not very strong, as it could simply mean we aren't smart enough to find those algorithms.

What we need is a more principled way of answering this question. When mathematicians are faced with an existential problem —in the sense of, does there exist an object with these properties, not in the sense of, you know, God is dead and all—what they do is try and find extreme cases that can represent the whole spectrum of objects to analyze.

In this case, we are dealing with finding problems that are difficult to solve. So it makes sense to ask, "What is the most difficult problem possible?" A problem so hard that if we crack it, we crack them all.

One problem to rule them all

That's the idea behind NP-completeness. Let's focus on these super tricky problems – the toughest ones in this field. If there are problems so tough that solving just one of them would also solve all the others, that would seriously challenge the idea of P equals NP.

These really tough problems are called **NP-complete** problems, a concept

defined by Stephen Cook in the 1970s. An NP-complete problem is basically a problem that is NP —that is, easy to verify—, and a solution to it in polynomial time would also give us a solution to any other problem in NP. So, in a way, these are the only problems we need to look at. If we solve one of these, we've proven P equals NP. And if we can't solve just one, we've proven P not equal to NP.

In short, we just need to focus on NP-complete problems. So, the main question then becomes: Are there problems so complex that they are essentially equivalent to all other problems?

Obviously, the hardest problems wouldn't revolve around everyday topics like finding paths in maps, sorting numbers, or solving equations. No, these problems should be much more abstract, so that they could encompass numerous other problems in their definition.

Cook's idea was to create a problem centered around problem-solving itself. For instance, how about *simulating a computer*? That would be an incredibly abstract problem. To make it even more challenging, they could turn this computer simulation into a decision problem —keep in mind that NP problems are decision problems.

One way to define this problem is to imagine having an electronic circuit that computes some logical formula —this is all computers do at their core, as all computable arithmetic can be reduced to logic. Let's take the simplest logical circuit, a completely stateless one, and ask: Is there any input that will make this circuit output True? If so, we call the circuit *satisfiable*. Otherwise, it is deemed *unsatisfiable*.

Hence, given an arbitrary logical circuit, the task of determining if it's satisfiable or not is called circuit satisfiability, or **Circuit-SAT** for short.

In 1970, Stephen Cook proved that Circuit-SAT is as hard or harder than all other NP problems. If you can solve circuit satisfiability, you can solve any other problem in NP. This means you can solve the traveling salesman problem and the knapsack problem, but also sorting, searching, and basically any easily verifiable problem.

The proof of this idea is quite involved and complex, and not something I can fully explain in this post, so I'll save that for a more detailed discussion later. But basically, since logical circuits are extremely expressive and powerful, and can compute almost everything, any decision problem in computer science can be transformed into a logical circuit that simulates it solution. Cook's proof actually involves constructing a circuit for an abstract NP problem, so it's pretty technical. But the intuition is that these things are basically computers, so you're just simulating any possible (decision) algorithm.

And voilá! This proves the existence of at least one NP-complete problem, meaning there is one problem in NP that is as difficult as all problems in NP. Stephen Cook's work in 1971 essentially kick-started the entire field of

computational complexity and defined the most important question in computer science: P versus NP.

The story, however, doesn't just end there. Circuit-SAT is great, but it is too much of an abstract problem. Just one year later, Richard Karp came along and demonstrated that 21 well-known and very concrete computer science problems were also NP-complete problems. These problems included the traveling salesman problem, the knapsack problem, various scheduling problems, and many graph coloring problems. In short, there turned out to be a whole bunch of problems that fell under this category, not just some abstract circuit simulation task.

These NP-complete problems aren't just theoretical issues, either. They are practical problems that we encounter regularly in logistics, optimization, and scheduling. After Karp proved his 21 original NP-complete problems, a wave of people started proving that nearly any problem in computer science involving combinatorics could be classified as NP-complete. As a result, there are now over 4,000 papers proving different NP-complete problems, ranging from determining the best move in Tetris to folding proteins.

This compelling evidence has led many computer scientists to believe that P != NP and that there are, indeed, problems that are fundamentally harder to solve, not just because we lack some understanding about them but because, by their very nature, they just cannot be solved efficiently.

What does this mean?

If it turns out that if P != NP, then there are some fundamental limits to how fast a computer can solve the majority of the most important problems in logistics, planning, simulation, etc. While we have many heuristics to solve either some cases perfectly or most cases approximately, all these problems may ultimately be unsolvable quickly.

Not even a superadvanced alien —or computer— could be exponentially faster than us. Thus, P vs NP might be our best weapon to stop a self-improving AGI from reaching superhuman capacity. Even gods can't escape complexity theory.

Formal language theory

Languages

Automata

Computational mathematics

Logic

Discrete math

Algorithms and Data Structures

Algorithms

What is an algorithm?

Searching and sorting

Linear search

Binary search

Naive sorting

Efficient sorting

Algorithm Design Strategies

Recursion

Greedy algorithms

Dynamic programming

Data Structures

Strings

String matching

Arrays

Array-based data structures

Lists

Queues

Stacks

Heaps

Trees

Graphs

Path finding

Computational Systems

Computer Architecture

Logical circuits

Microprocessors

Memory

Operating Systems

Computer Networks

The TCP-IP Protocol

Distributed Systems

Cloud Computing

Cryptography

Software Engineering

Programming Languages

Principles of Software Design

Databases and Information Systems

Software Platforms

Desktop software

Mobile software

The World Wide Web

Human-Computer Interaction

Videogames

Artificial Intelligence

Knowledge representation and reasoning

Logical reasoning

Ontologies

Search

Informed search

Adversarial search

Constraing satfisaction

Metaheursitics

Machine Learning

Learning paradigms

Supervised learning

Unsupervised learning

Reinforcement learning

Applications

Tabular data

Signal processing

Computer Vision

Natural Language Processing

Large Language Models

Epilogue

The Road Ahead