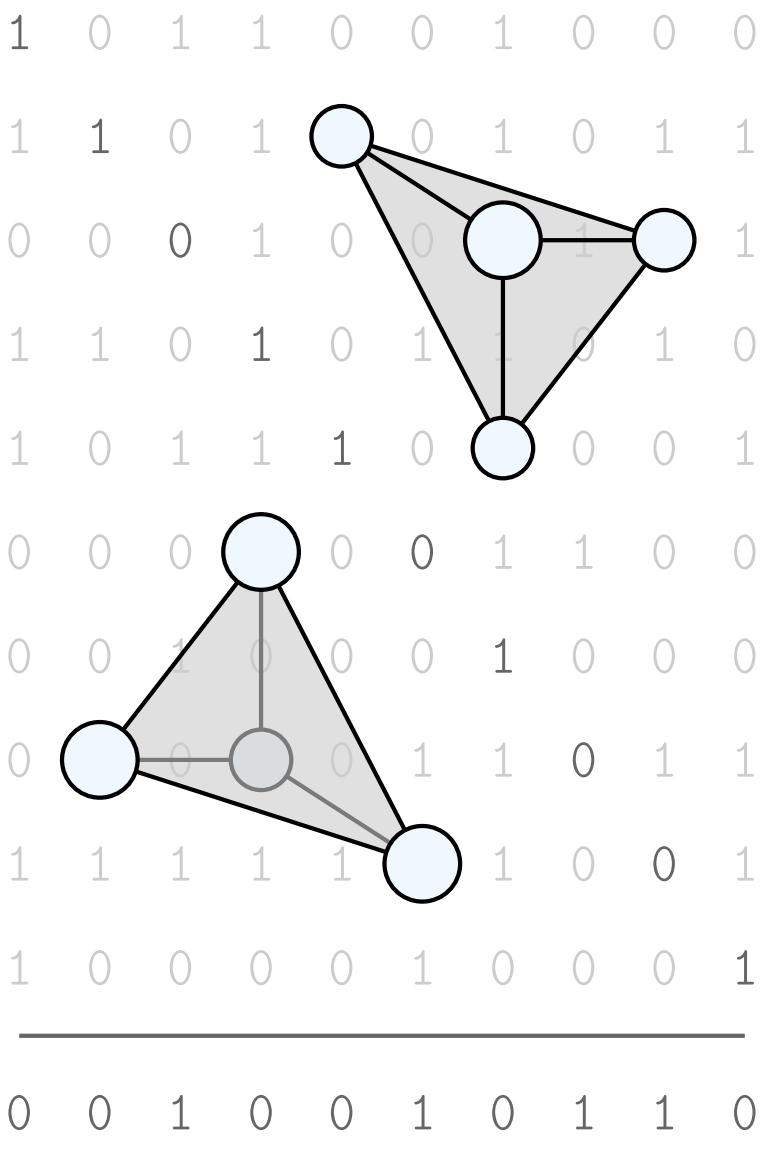
INTUITION FOR COMPUTATION

A Guide for Programmers and Curious Minds



Tommy Monson

Intuition For Computation

Contents

The very word intuition has to be understood.

You know the word tuition—tuition comes from outside, somebody teaches you, the tutor.

Intuition means something that arises within your being;

it is your potential, that's why it is called intuition.

Wisdom is never borrowed, and that which is borrowed is never wisdom.

Unless you have your own wisdom, your own vision,

your own clarity, your own eyes to see,

you will not be able to understand the mystery of existence.

—Osho

Preface

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

—Toni Morrison

Acknowledgements

Introduction

> *So many people today—and even professional scientists—seem to me like somebody who has seen thousands of trees but has never seen a forest. A knowledge of the historic and philosophical background gives that kind of independence from prejudices of his generation from which most scientists are suffering. This independence created by philosophical insight is—in my opinion—the mark of distinction between a mere artisan or specialist and a real seeker after truth.*
>
> —Albert Einstein

I find it difficult to describe what this book is. And yet, I find it easy to express why I think it is worth your time and contemplation.

Part I.

Philosophy of Computation

> *We're presently in the midst of a third intellectual revolution. The first came with Newton: the planets obey physical laws. The second came with Darwin: biology obeys genetic laws. In today's third revolution, we're coming to realize that even minds and societies emerge from interacting laws that can be regarded as computations.* ***Everything is a computation.***
>
> —Rudy Rucker

Computation is an essential part of being human. Just as we act on our perceptions, feel our emotions, and daydream in our imaginations, we also compute with our *reason* in order to find answers to the questions we have about life.

One can describe computation in a variety of ways. Some are poetic, others are more formal, and many we will discuss at length. In doing so, we will generally flow from poetry to formality, starting with broad statements and colorful examples and progressing toward a more sophisticated understanding. For now, we will say that computation is a process which resolves *uncertainty*.

In his *Theogony*, the Ancient Greek poet Hesiod (fl. 750 BCE) depicts a fascinating uncertainty called Chaos: a vast nothingness that preceded the creation of the Universe. From this void emerged the primordial deities, personifications of nature who gave life to the Titans and, by extension, to the Olympian gods. Curiously, the modern definition of chaos has nearly inverted. Now, one might say that a situation is chaotic if there is *too much* going on. Despite their great conceptual difference, these meanings share a common property: they both evoke *confusion*.

This place in which we find ourselves may indeed be post-Chaos, but confusion remains a familiar state for us. In fact, it is our natural state. We come into life confused about everything—crying, open-eyed in reaction to how much is going on around us. Slowly, we figure a few things out. Later, we consider many things *certain*. As time ticks on and we carve our separate paths in the world, the wise ones among us come to realize the true depth of this confusion.

Although we are initially confused, we naturally pursue an understanding of the chaos that surrounds us. We perceive the goings-on of our environment and organize them into patterns that orient us and give meaning to the disorder of our existence. This is the crux of computation. It is the structured manipulation of *data* into *information* for the purpose of acquiring *knowledge*. It shines a light into the darkness we call home.

These terms—*data*, *information*, and *knowledge*—are broad enough to apply to a variety of systems. In fact, they appear in nearly every academic field and certainly within those that strive for objectivity. One can have data on, information about, and knowledge of anything that can be observed or experienced. As such, it is difficult to confine these terms to short and tidy definitions that are both rigorous and free of controversy. We will have to settle then for a long, pragmatic discussion.

Still, we must start somewhere. For example, you already have an idea of what information is, and it is likely that your intuition is broadly correct: information is a thing that tells you something. As we dive into more and more abstract topics, do not hesitate to, in this way, fall back on your existing vocabulary for support. A word used technically is more precise than the same word used casually, but the gist of it all is often the same. Or, at least, there is often some sense in which one reflects the other, like concepts linked

in a good simile.

Recall the usual context in which technical language is born. A *pattern* or *phenomenon* is observed by a community, and it becomes the subject of casual discourse. It may even be a widespread concept that is discussed by society at large. Accordingly, the phenomenon is given an *informal definition* and a *name* (or, perhaps, many names). Interested theoreticians then formalize it; they describe its *form* in unambiguous terms. They give it a *formal definition* and bestow a more permanent name upon it, either one used previously or something new that, in the namers' opinion, better conveys the idea. In the latter case, the new term is disseminated throughout the community and perhaps to the wider public, where it may either replace or coexist with previous informal terms. Regardless, the namers must reach outside of the formalisms of their field to find a word from an informal, natural language that captures the essence of their abstract thought.

This is the beauty of a good piece of jargon: such a word will, to an amateur, shed a shallow but wide light into the vast space of an idea, but it will also illuminate the whole shebang in the eyes of an expert, evoking in its concise informality all that he or she has previously learned on the subject. In the spirit of this principle, our three words are defined softly below to act now as beacons in the dark of our ignorance and later as shining summaries of our hard-earned understanding:

> **Data** (sing. **datum**) are pieces of information that are meaningless when considered separately. They are observable lacks of uniformity in reality, each of which may be represented by a symbol that conveys its particular difference from the norm.
>
> **Information** is an ordered arrangement of data that has the potential to convey something meaningful. The data are arranged such that the information as a whole complies with the grammar of a language that can, in theory, be understood.
>
> **Knowledge** is an understanding of information that emerges given proper interpretation and purposeful thought. It is usually associated with truth and may be considered a body of true beliefs.

The journey to an honest appreciation of these terms will be an arduous one, and subtly so, as it will require an open mind and an ample store of time set aside for the contemplation. Nevertheless, walking that inner path is unquestionably worth the cost when one considers how pervasive these concepts truly are. They can describe any event, including those that occur in computers as they run their software and in brains as they continuously perceive and conceive of their own respective conscious experiences. This chapter explores Man's longstanding interest in these scraps of reality: our inquiry into the smallest details of Nature; our concretization of abstract thought through language, signage, and code; our distillation of human reason into mechanical rulesets.

There is a fourth word—*wisdom*—but it is even harder to pin down than the previous three. It is a lofty word—so lofty that it is difficult to say anything about it without sounding like a proselytizer. Its utterance conjures up images of ashen-haired, big-bearded men in togas or robes, speaking of grandiose things or passing time in peaceful admiration of the world. It has no formal definition because its form is unknown, close enough to touch momentarily but always out of reach. Furthermore, it is bound to messy human

concerns like emotions, morals, and purpose. How, then, could wisdom be relevant to the study of something as well-defined as computation?

1. Cognitive Ontology

> *The first rule is to keep an untroubled spirit. The second is to look things in the face and know them for what they are.*
>
> —Marcus Aurelius

Dark turns into light. A boundless zephyr of blurs. You see. You hear. You *feel*. The knife-edge of experience materializes, and you step out onto it. It cuts you, and there is no thought—only sensation. Air kissing skin and turbulent color. Sonic frenzy and the feeling of self-weight. Reality emerges raw and beams through all you are. You are lost in it all: a sailboat in a storm, a satellite adrift in space, and the rest. Indifferent, it rages on.

This is roughly the experience of a newborn infant. For children of a month or so, life is a sensory mayhem of little sense. They are not conscious, at least not in the way that we typically think of consciousness. They are, however, aware of their surroundings and sensitive to *phenomena* (i.e. the *appearances* of reality, *observables*). To newborns, the world is not a system of distinct parts but an all-encompassing soup of stimuli. Their sensory organs input data that they do not understand, and they then output *reflexes* honed through millions of years of natural selection.

This is not to say that babies are mindless. Rather, their minds are just unlike those of typical adults, which both *filter* and *store* information. For example, while babies and adults have similar hearing ability, babies nevertheless hear things that adults do not. This is because adults subconsciously ignore certain sounds; they may *sense* the auditory data, but they do not *perceive* them. This filtering process is called *sensory gating*, and it inhibits any stimulus that is deemed irrelevant. Any information that is not filtered is then eligible for storage in *memory*.

Thus, if the adult mind is like the conical beam of a flashlight, seeing far but neglecting much, the baby mind is like the radiant orb of a lantern, seeing all that is immediate but illuminating nothing deeper. These configurations are well-suited to the goals of each: the adult needs *attention* to understand complex situations whereas the baby needs *unrestricted perception* to acquire as much information as possible. Unburdened by categorical thought and sensory gating, babies live in the present and see the suchness of their surroundings. However, their experience is not quite the *kenshō* of Zen—it is more like Aldous Huxley's Mind at Large or William Blake's oft-quoted metaphor, from which Huxley took the title of his first essay on psychonautics:

> If the doors of perception were cleansed every thing would appear to man as it is: Infinite. For man has closed himself up, till he sees all things thro' narrow chinks of his cavern.

So it is through a great deal of concurrent filtering and learning that we come to build a cavern around ourselves. But we do not muffle our perception pointlessly—we trade it for genuine *thought*.

1.1. Upper Ontology

There is a distinction that is sometimes made in cognitive science between a *(phenomenal) P-consciousness* and an *(access) A-consciousness*, the former being the sole consciousness of the baby mind, concerned only with bits of immediate, subjective experience known as *qualia* (e.g. what it is like to *feel* a delicate raspberry as it touches your tongue, what it is like to *taste* it as you mash it between your molars, the rush of brief and unique *delight* found in enjoying that *particular* berry, etc.) and the latter being the dominant consciousness of the adult mind, concerned with *cognitive information* (e.g. thoughts about sensory data, abstract ideas, memories of the past, plans for the future, and all things involved in *mental computation*).

Of course, adults still have a P-consciousness—we can lose ourselves in the moment if we are willing. But ours is a boarded-up P-consciousness, largely unaware of the immense volume of all that is happening at all times. Perhaps it is only the bodhisattvas or the mystics of yore who truly come to know their infant minds again and see the P and A as they are: two *modes* of a unified whole. Or perhaps, more pessimistically (and as Huxley suggests), the P in its unadulterated totality is really a kind of schizophrenic insanity, and it is only by the grace of our specialized neural hardware that we erect defenses against its sensory onslaught. Alas, the suchness of such a consciousness eludes those of us with at least one foot planted solidly on the earth.

1.1.1. The Hard Problem of Consciousness

What is it about phenomenal consciousness that is so intangible? It has been a question of interest to humanity for thousands of years and an object of formal study since René Descartes (1596-1650) first posited the duality of *mind* and *body*. Inspired by the proliferation of *automata* (i.e. self-operating machines) in Paris, Descartes came to suggest the extraordinary: that animals—their limbs, organs, and mannerisms—could be replicated with sufficiently complex machinery. Further, he professed that animals *themselves* were machines, with bone and flesh standing in for wood and metal:

> It seems reasonable since art copies nature, and men can make various automata which move without thought, that nature should produce its own automata much more splendid than the artificial ones. These natural automata are the animals.

Note that Descartes implies here that animals are "without thought." He also claimed that thoughts required a language in which rational ideas could be expressed. (Thoughts encoded in a language, however, need not be expressed outwardly; babies, for example, mentally represent ideas before learning how to render them in speech.) He also reasoned on this basis that an automaton would never be able to think because it would never be able to speak in the way that humans do—by organically producing an appropriate response to any given prompt. (Whether or not he was correct remains to be seen,

though the possibility of an *artificial general intelligence* does not seem quite as absurd now as it must have seemed back then.) In Descartes' view, humans are special and cannot be reduced to machinery because, unlike animals, they possess rational minds enriched with a free will that no algorithm can replicate.

Thus, Descartes championed *substance dualism* (also known as *Cartesian dualism*), in which all things were either fundamentally of *matter (res extensa)* or of immaterial *mind (res cogitans)*. He saw the human being as a union of the two disparate substances but struggled to reconcile his strict dualist views with the decidedly hybrid and experiential character of his own bodily sensations and emotions. In the sixth and final meditation of his *Meditations on First Philosophy*, Descartes states that in being skeptical of his fallible senses, he understands himself to be essentially a mind, a purely thinking thing of no physical extent that is intuitively indivisible. And yet, he nevertheless *feels* the qualia of his body and thus must straddle his own *mind-body duality*, being, in a phrase, both flesh and not:

> There is nothing which this nature teaches me more expressly [nor more sensibly] than that I have a body which is adversely affected when I feel pain, which has need of food or drink when I experience the feelings of hunger and thirst, and so on; nor can I doubt there being some truth in all this.
>
> Nature also teaches me by these sensations of pain, hunger, thirst, etc., that I am not only lodged in my body as a pilot in a vessel, but that I am very closely united to it, and so to speak so intermingled with it that I seem to compose with it one whole. For if that were not the case, when my body is hurt, I, who am merely a thinking thing, should not feel pain, for I should perceive this wound by the understanding only, just as the sailor perceives by sight when something is damaged in his vessel; and when my body has need of drink or food, I should clearly understand the fact without being warned of it by confused feelings of hunger and thirst. For all these sensations of hunger, thirst, pain, etc. are in truth none other than certain confused modes of thought which are produced by the union and apparent intermingling of mind and body.

Fast-forward to the turn of the 20^{th} century and general confidence in the scientific method had annealed Descartes' mechanistic philosophy into full-blown metaphysical naturalism. The Modernist movement was in full swing, quantum mechanics was being formulated, mathematics was being axiomatized, and about thirty years later the greatest minds of the era would contribute to a general theory of computation. And in the 1950s, the *cognitive revolution* began, and people started to think of the mind as a complex system that could be explained in terms of information and computation. It seemed like the next logical step in human progress. Like matter had been reduced to elementary particles, like water had been reduced to H_2O, like genes had been reduced to DNA, so too would the mind be reduced to its constituent parts.

And yet, there is also that deep feeling in us that the mind is something else. We are inclined to believe that there is something about consciousness that is incomparable to, say, a Rube Goldberg machine or a computer running a program, no matter how complex either may be. Namely, we feel that there is *something that it is like to experience being*. In 1974, philosopher of mind Thomas Nagel (b. 1937) brought this idea to the attention of the burgeoning field of *cognitive science* with his paper *What Is It Like to Be*

a Bat?, in which he states that no physicalist theory will capture the essence of the mind until we come to understand its subjective aspects:

> The fact that an organism has conscious experience at all means, basically, that there is something it is like to be that organism. There may be further implications about the form of the experience; there may even (though I doubt it) be implications about the behavior of the organism. But fundamentally an organism has conscious mental states if and only if there is something that it is like to *be* that organism—something it is like *for* the organism. We may call this the subjective character of experience.

This "subjective character of experience" presents a major challenge to the belief that neuroscience will eventually reduce consciousness to a theory. The *philosophy of mind* is then, perhaps, brazenly Postmodernist—inherently subjective and relativistic, with answers that will remain unknown to mankind in spite of its scientific progress. Nagel posits, for example, that we cannot conceive of the subjective experience of a bat because it is totally alien to our own experience. And further, he argues that it does no good to ground such an experience in greater and greater *objectivity* (i.e. independence from human bias), that such first-person character is stripped away in the third-person framework of science:

> It will not help to try to imagine that one has webbing on one's arms, which enables one to fly around at dusk and dawn catching insects in one's mouth; that one has very poor vision, and perceives the surrounding world by a system of reflected high-frequency sound signals; and that one spends the day hanging upside down by one's feet in an attic. In so far as I can imagine this (which is not very far), it tells me only what it would be like for *me* to behave as a bat behaves. But that is not the question. I want to know what it is like for a *bat* to be a bat.
>
> . . .
>
> If the subjective character of experience is fully comprehensible only from one point of view, then any shift to greater objectivity—that is, less attachment to a specific viewpoint—does not take us nearer to the real nature of the phenomenon: it takes us farther away from it.

For Nagel, this is not necessarily a death knell for any formal understanding of consciousness, but it *is* a declaration of doubt in the capacity of science *as we currently know it* to shed light on such matters. His qualms lie not with physicalism itself—that is, not with the assertion that mental states are fundamentally physical—but with an overconfidence that tends to come with the territory of such a mindset. His goal is to elucidate a problem: that, while it is reasonable to believe that one's mind is the result of solely the neurobiological mechanisms in one's body, it is nevertheless *unreasonable*, if we accept that qualia exist and are essential to our experience, to claim that science is presently capable of capturing the subjective character of the conscious mind within its objective bounds.

In light of this issue, Nagel concludes with a call for the formulation of an "objective phenomenology," a framework in which qualia could be described independently of

their experiencer's point of view. Only then could our subjective character of experience, which is so central to our conception of being human, be expressed "in a form comprehensible to beings incapable of having those experiences." As it stands, we cannot describe our seeing of red without analogy to previous experiences of seeing red, and in such terms we can only convey our meaning to those with eyes and brains like our own. And if bats could talk, they would be similarly unable to communicate their echolocational qualia to our sonar-ignorant minds. Thus, *What Is It Like to Be a Bat?* is not a position piece on the nature of qualia but a declaration of agnosticism toward it: we cannot say anything objective about what we feel until we determine whether or not our feelings have objective content.

Nevertheless, Nagel is often considered a standard-bearer for *property dualism*, which holds that there are two distinct kinds of properties: *physical* and *mental*. Unlike Cartesian dualism, this position holds that there is only one kind of substance, and, in contemporary Western philosophy, it is almost always considered physical rather than mental, an objective entity rather than a subjective one. But Nagel is particularly a standard-bearer for a variety of property dualism known as *neutral monism*, which offers a middle path: that everything is composed of a neutral stuff that is neither physical nor mental. An early form of this view was put forth by the pragmatic philosopher William James (1842-1910) in his essay *Does Consciousness Exist?*:

> My thesis is that if we start with the supposition that there is only one primal stuff or material in the world, a stuff of which everything is composed, and if we call that stuff 'pure experience,' the knowing can easily be explained as a particular sort of relation towards one another into which portions of pure experience may enter. The relation itself is a part of pure experience; one of its 'terms' becomes the subject or bearer of the knowledge, the knower, the other becomes the object known.

So there is a particular relation $\rightarrow$, one among many, and it is called *knowing*. And it relates one term A to another term B. And we say that A, the knower, *knows* B, the known: $A \rightarrow B$. And all of these—the $\rightarrow$, the A, the B, and the composed whole—are of a neutral, experiential substance. And, indeed, so it is for everything that we may perceive or conceive. The onus is placed then on the rational interpreter to determine whether a relational structure (or *pattern*) of neutral elements constitutes a physical or mental property. Thus, we are pattern-scanning machines and arbiters of what is *thought* and what is *thing*.

Regardless of whether or not they exist, qualia are beyond our scope.

1.1.2. The Categories of the Mind

Space and time are two features of our reality. As they appear to us, the former is an expanse in three dimensions, and the latter can be thought of as an arrow that extends steadily in one dimension, from past to future. They are inextricably woven into the fabric of our experience, and yet their true nature eludes us.

Consider a space that contains many particles of *substance*, all of which move about constantly as time goes on. Suppose that, one by one and randomly, these particles blink out of existence forever. When only one particle remains, is that which is *not*

the particle still considered space? Further, can we still say that this particle moves? It remains surrounded by a void, interacting and relating with nothing at all, participating in no distinguishable events. And if indeed nothing changes, does time still tick? If so, will space and time *mean* anything after that last, lonely bit of matter disappears and everything becomes a uniform non-existence?

These are the questions that pertain to whether space and time are *absolute* or *relative*. If space is absolute, like a fixed three-dimensional grid that pervades the material realm, we might ask where the origin $(0, 0, 0)$ is. If such a location were to exist, every object and particle in the Universe would have an absolute position in three coordinates. In other words, position would be a distinguishing feature of each and every object, and it would not be defined in relation to other objects, but to this primordial *center of the Universe*.

If space is instead relative, a conceptual origin can be placed wherever one likes. Distances can then be measured from this point. Of course, this can be done in an absolute space as well. Thus, the absolutist viewpoint commits the cardinal sins of being *unobservable* and being *useless*. To any human observer, a reality with absolute space would look no different than a reality with relative space—objects would not change in any way after being related to an arbitrary center. Access to these absolute positions also would not provide us with any additional information—they would just be fixed offsets of any positions measured from a relative point.

Perhaps then, space is is better thought of as *relational* rather than *objective*. Perhaps it is not a hidden grid-like object itself, but an *ordering* that is composed of relations between objects. And perhaps then it is also better thought of as a product of the *mind* rather than as a physical thing. That is, space may instead be an *abstraction* that our minds make by estimating the relative distance between two objects and comparing it to every other potential spatial relation in our surroundings. In this view, space is simply a web of relations that is constructed by a sentient being.

Similarly, time may not be a hidden clock-like object, but instead a series of relative temporal intervals between *events*. And if we only know that time passes because of the changes we perceive in space, it is possible that time is an abstraction as well, an ordering of events made by a sufficiently conscious mind. Furthermore, if space and time are both relational and coupled, perhaps it is better to think of them as a unified *spacetime* despite our predispositions to think of them separately.

The philosophy of space and time is full of eternal questions about infinity, substance, void, uniformity, asymmetry, and consciousness. Comparatively, *computational space and time* are simple.

This curiosity aside, we could also design our category system such that quantities and qualities are both considered *properties* of some very general sort of thing. Philosophers have long considered the idea of a highest-level category and have ascribed a variety of names to its members: *entity*, *being*, *thing*, *term*, *individual*, etc. Each carries its own subtle connotations. We will use the name *object* because it is the default choice in current philosophical practice and because it has bled into the technical language of computer science with its meaning largely unchanged.

How general is this category of objects? Namely, are properties objects? If they are, then our system collapses to a single category, which really means that no distinctions

about reality can be made at all. The category of objects thus becomes the soupy mess we are trying to escape. For now, let us assume that properties are *not* objects. Instead, they are something else, and they *relate* to objects in some way, giving them their characteristics. From these relations, the diversity of existence unfolds.

The **abstract-concrete dichotomy** classifies *things* as being either *abstract* or *concrete*. Despite the fact that most people recognize a difference between an abstract and a concrete thing, this dichotomy does not have a universally accepted definition. That said, being abstract is often defined as being both *non-spatiotemporal* and *causally inefficacious*. That is, something is abstract if and only if

1. it does not exist in *space* or in *time* (or, alternatively, in *spacetime*), and
2. it cannot be the *cause* of any *effect*.

However, one might argue that nothing is truly causally inefficacious. Indeed, if something was truly unable to cause *anything*, it would not be able to affect the *thoughts* of any philosopher, mathematician, or scientist for whom such an abstract thing would be of interest. And yet *mathematical objects* (e.g. a number, a line, a cube, the sine function, etc.) are often considered abstract, despite the fact that they affect the thoughts of mathematicians daily, thoughts which in turn prompt concrete behavior, such as drawing diagrams. Perhaps, causal inefficacy is too strict a requirement.

Let us consider why one might want to refer to something that is abstract. An abstract object references *properties* that many concrete, causally efficacious objects have. For example, a sphere does not exist in space or time; it is *nowhere* and *never*. It also does not seem capable of causing an effect in the way that, say, a ping-pong ball can. And while the abstract sphere may not have the same sort of causal efficacy as concrete spherical objects, it does have a *relation* to those objects. That is, the concept of sphericality is *related* to the *class* of concrete objects that may be reasonably modeled as spheres. Thus, an abstract object is, at least, convenient because it can identify a class of concrete objects that is of interest by specifying relevant properties.

In light of recent advancements in fields like neuroscience and artificial intelligence, many believe that the human mind is nothing more than a physical system, albeit a particularly complicated one, that can be modeled mathematically like any other. So it may be that abstract objects are just interpretations of concrete electrochemical signals in our brains. Regardless,

A common approach to this challenge is creating a system of *categories* that partition reality at the highest level. For example, one could suggest that there is a fundamental difference between a *quantity* and a *quality*. If these are considered distinct categories, we can classify *measurable* or *countable* stuff like *length*, *mass*, *number*, and *monetary value* as quantitative and *experiential* stuff like *softness*, *roundness*, *flavor*, and *beauty* as qualitative.

The quantity-quality distinction is a generally accepted one. However, the dichotomy is not so absolute. For example, softness is listed above as a quality. We might expect, then, that hardness is a quality. For example, rocks are hard and puppy ears are soft. But hardness is also a quantitative measurement of resistance to plastic deformation. Thus, the word *hardness* can refer to both quantitative and qualitative phenomena. The categorical difference between these hardnesses, which exists independent of language, is identified through the context in which the word is used. A hardness can be measured

with numbers, but it can also simply be *experienced* via touch.

A **domain of discourse** is a set of all the things under discussion. It is also called a *universe*, particularly in the field of mathematical logic. It specifies, out of all conceivable objects of study, those objects which are pertinent to the matter at hand. It gives *context* to what we say.

A *discourse* is a conversation, and its *domain* is the subject of the conversation. A domain may also refer to an entire field of study, perhaps one whose conversation has been developing for thousands of years (e.g. physics, whose domain is physical reality). Each field of study has an accompanying domain of discourse, the set of objects within its purview. Working in such a field is thus akin to exploring a *universe* populated with such objects.

As we observe the objects of a universe, we can make *logical statements* about them. We can also ensure our statements are consistent by using a *formal system of deduction* to prove them from a set of *axioms*. And if we compile every statement that can be proven in this system—the axioms and their consequences—we will form a *theory*, a set of true statements (or *theorems*) that describes the universe in question.

The word *theory* may also refer to a *theory-in-progress*, a body of logical work to which theoreticians contribute. We might consider this an *open theory*, a theory for which we do not have all possible knowledge. In contrast, a *closed theory* is one for which we know everything there is to know.

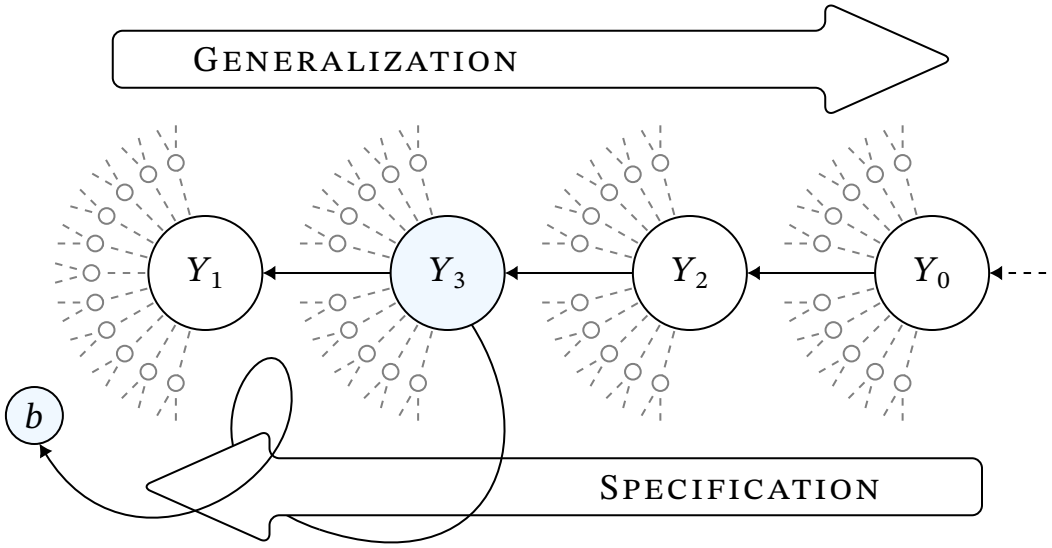
Similarly, we can describe universes as open or closed. The *open-world assumption* is made when we do not have complete knowledge of a system: if a statement is not known to be true (i.e. it is not a member of our theory), we do not assume anything about its truth value. This is the case for any system that is *discovered*, like the various systems of nature.

Computational systems, however, are not discovered but *designed*. Under normal conditions, it is possible to know everything that happens during a digital computation—all of the information appears in a discrete, finite space of computer memory. Thus, we are omniscient with regard to this universe and should make the *closed-world assumption*: if a statement is not known to be true, it must be false.

The **type-token distinction**

Identity seems to be a mandatory quirk of being. It is difficult to imagine an existence in which things are not themselves. And yet, it is also difficult to define, in any meaningful way, this supposedly crucial concept.

The **map-territory relation**


GENERALIZATION
Y_1
Y_3
Y_2
Y_0
b
SPECIFICATION

2. Metaphysics, Natural Philosophy, and Physics

I just wondered how things were put together.

—Claude Shannon

2.1. Classical Metaphysics

2.1.1. Mythology and Cosmogomy

2.1.2. Pre-Socratic Thought: *Henosis* and *Arche*

2.1.3. Democritus' Atomism

2.1.4. Plato's Theory of Forms

Plato (c. 428/427 – c. 348/347 BCE) was the first to expound a metaphysics of objects with his *theory of Forms*, which proposed a solution to the *problem of universals*. This problem is subtle, but important: how can one general property appear in many individual objects? For example, when we say that a floor and a bowl are *smooth*, are we referring to some singular paradigm of *smoothness* that is independent of each? For Plato, *universals* or *Forms*, such as smoothness, are indeed distinct from the *particulars* that *partake* of them, such as floors and bowls and other smooth objects. He posits that the *essence* of an object, its *sine qua non*, is the result of its partaking of these Forms.

Forms, as conceptualized by Plato, are ideal, abstract entities that exist in the *hyperouranos*, the realm beyond the *heavens*. In the times of Homer and Hesiod, the Ancient Greeks understood a cosmology that closely mirrored that of the Hebrew Bible. They envisioned the Earth as a flat disc surrounded by endless ocean and the sky as a solid dome with gem-like stars embedded in it, much as it all appears from a naive perspective on the ground. Under the Earth was the *underworld*, where the dead live, and beyond the dome were the *heavens*, where the divine live.

Plato, however, was influenced by the philosophy of Pythagoras (c. 570 – c. 495 BCE), who championed a spherical Earth for aesthetic reasons. Thus, he instead modeled the Earth as a "round body in the center of the heavens," and he imagined the Forms as part of a different, eternal world that transcends physical space and time. Whereas physical reality is the domain of perception and opinion, the realm of the Forms is, as he paints it in *Phaedrus*, the domain of *reason*:

> Of that place beyond the heavens none of our earthly poets has yet sung, and none shall sing worthily. But this is the manner of it, for assuredly we must be bold to speak what is true, above all when our discourse is upon truth.

> It is there that true Being dwells, without colour or shape, that cannot be touched; reason alone, the soul's pilot, can behold it, and all true knowledge is knowledge thereof. Now even as the mind of a god is nourished by reason and knowledge, so also is it with every soul that has a care to receive her proper food; wherefore when at last she has beheld Being she is well content, and contemplating truth she is nourished and prospers, until the heaven's revolution brings her back full circle. And while she is borne round she discerns justice, its very self, and likewise temperance, and knowledge, not the knowledge that is neighbor to Becoming and varies with the various objects to which we commonly ascribe being, but the veritable knowledge of Being that veritably is. And when she has contemplated likewise and feasted upon all else that has true being, she descends again within the heavens and comes back home. And having so come, her charioteer sets his steeds at their manger, and puts ambrosia before them and draught of nectar to drink withal.

Plato makes a revolutionary statement here about the nature of scientific knowledge. He declares that one cannot know the physical world directly because it is, in fact, *less real* than the *hyperouranos*. As he puts it in *Timaeus*, a Form is "what always is and never becomes," and a particular is "what becomes and never is." Further, physical reality is merely the "receptacle of all Becoming," a three-dimensional *space* that affords spatial *extension* and *location* to all particulars. Unlike Forms, particulars are subject to *change*, and thus cannot be sources of absolute knowledge. They are slippery, metaphysically undefinable, and explainable only in terms of the unchanging Forms of which they partake.

The above passage can be interpreted as an early argument for basing human knowledge in abstract mathematics rather than in subjective evaluations of concrete observations. For example, to *know* what a triangle is, one cannot simply point to a triangular rock or even to a very carefully drawn diagram of a triangle. One must instead use *abstract reasoning* to express the Form of a triangle by means of a *formal definition*: a three-sided shape in a two-dimensional plane. This is the *essence* of a triangle. In contrast, the triangular rock and diagram each partake of multiple Forms, and neither are perfectly triangular. We may still *call* them triangles, but only those who know the Form of a triangle will understand what we mean when we do so. Therefore, it is knowledge of the unambiguous Form that gives insight into the complex and ever-changing particular, not the other way around.

Here, our object-property distinction blurs. Because while Forms may define universal properties, are they not also objects in an abstract sense? And while concrete particulars may be objects, are they not also fully characterized by a set of properties? In Plato's metaphysics, the categories of interest are really the *concrete* and the *abstract*, that which we *perceive* and that which we *conceive*. Plato further claims that humans come from the abstract realm, where they familiarize themselves with the Forms, and they are born into the concrete realm, where particulars are abundant and Forms are only latent memories. It is through philosophy then, the exploration of the rational mind, that we unearth our innate understanding.

To explain the link between the realms, Plato also postulates a *demiurge*, a transcendent artisan who shapes an initially chaotic space into an ordered, material reality, using the Forms as models. This entity is benevolent and wishes to craft a world like the ideal one he sees, but the objects that he *instantiates* are always imperfect copies. Like that of

a master portraitist, his work is beautiful, but it is, in the end, a mere *image* of his subject.

So it is for programmers. Before us lies the space of computer memory, uniform and featureless, a receptacle of Becoming. Our Forms are *types*, our particulars are *values*, and the behavior in the space is dictated by the *code* that we execute. In *object-oriented programming* (OOP), values can be of an *object type* that is formally defined by a *class*.

For example, my cat Mystic belongs to the class of entities known as Cat, the set of all possible cats. She is an *object* known as a cat, and her catness is defined by the class Cat. In Plato's terms, she is a *particular*, partaking of the Form *Cat*. Just as the demiurge instantiates particulars from Forms, so the programmer instantiates objects from classes. Both bridge the gap between what is possible and what is.

That said, while Plato's metaphysics gives good insight into OOP and other abstract (or *high-level*) programming concepts, it cannot explain concrete (or *low-level*) ideas like *data* that is represented with *bits*. This is because Plato was not overly concerned with the *substance* of objects, the stuff that makes them *physical*. Thus, in our pursuit of a holistic understanding of *information*, we must also consider *substance theories*, philosophies that give metaphysical weight to *matter*.

2.1.5. Aristotle's Hylomorphism

2.2. Theories and Models

2.2.1. Euclid's *Elements*

Although arithmetic has been practiced since prehistory, theoretical mathematics was founded by the Ancient Greeks. The Greeks, in contrast to earlier peoples, applied *deductive reasoning* to the study of mathematics. While sophisticated arithmetic, geometry, and algebra was done before this period by the Babylonians and the Egyptians, this work was all based on *evidence* collected from the physical world. The Greeks were the first to recognize the need for *proof* of mathematical statements using *logic*.

While various proofs were written by Greeks in the centuries before his birth, Euclid of Alexandria is nonetheless credited with formalizing the concept of proof due to his use of the *axiomatic method* in his groundbreaking treatise, *Elements*. This work was the first of its kind, serving as both a comprehensive compilation of Greek mathematical knowledge and as an introductory text suitable for students. Additionally, it was a product of a synthesis of Greek thought, founded on the epistemology and metaphysics of Plato and Aristotle.

Elements is divided into thirteen books. Books I–IV and VI cover *plane geometry*, the study of shape, size, and position in two-dimensional space. Books V and VII–X cover elementary *number theory* (classically known as *arithmetic*) in a geometric way, expressing numbers as lengths of line segments. Finally, Books XI–XIII cover *solid geometry* by applying principles of plane geometry to three-dimensional space. Each book begins with a list of *definitions* (if necessary) that assign names to abstract mathematical ideas that are expressed in unambiguous terms. These ideas are then used to prove *propositions*, mathematical statements that warrant proof.

Euclid's proofs in *Elements* can be described as *axiomatic*. That is, they start from *axioms*, statements that are taken as true without proof. Axioms solve a problem in

logical argumentation known as *infinite regress*, which often occurs, for example, when children ask questions. A curious child, seeking knowledge, might ask his parent why something is the way it is. That is, he requests a proof of some proposition P_0. The parent would respond with some proposition P_1 that supports the truth of P_0. The child, unsatisfied, would subsequently ask why P_1 is true. The parent would respond with the assertion that P_2 is true and implies P_1, and the child would similarly question whence P_2 arises. This dialogue could continue in this manner forever with the child endlessly searching for an anchor with which he can ground his understanding. However, patience waning, the parent must end this line of questioning by giving an unquestionable answer. Common examples include "Because that's just the way it is" or "Because I said so."

Some Greek philosophers considered infinite regress to be a serious epistemological issue. If we assume that all knowledge must be demonstrable by a proof or logical argument, we cannot know anything because our axioms cannot be satisfactorily proven. And yet, there are things that we claim to know. In his *Posterior Analytics*, Aristotle challenges this notion that all knowledge must be provable, averring that

> All instruction given or received by way of argument proceeds from pre-existent knowledge.

Aristotle continues his text in support of this statement. He discusses the nature of *premises* in his *syllogistic logic* and how they differ from the *conclusions* that they derive. Namely, he states that the premises of a knowledge-producing deductive argument or *syllogism* must be *true*, to ensure the truth of their conclusion, *indemonstrable*, to ensure that they are independent of proof, and *causal*, to ensure that their conclusion logically follows:

> Assuming then that my thesis as to the nature of scientific knowledge is correct, the premises of demonstrated knowledge must be true, primary, immediate, better known than and prior to the conclusion, which is further related to them as effect to cause. Unless these conditions are satisfied, the basic truths will not be 'appropriate' to the conclusion. Syllogism there may indeed be without these conditions, but such syllogism, not being productive of scientific knowledge, will not be demonstration. The premises must be true: for that which is non-existent cannot be known—we cannot know, e.g. that the diagonal of a square is commensurate with its side. The premises must be primary and indemonstrable; otherwise they will require demonstration in order to be known, since to have knowledge, if it be not accidental knowledge, of things which are demonstrable, means precisely to have a demonstration of them. The premises must be causes, since we posess scientific knowledge of a thing only when we know its cause; prior, in order to be causes; antecedently known, this antecedent knowledge being not our mere understanding of the meaning, but knowledge of the fact as well.

Thus, Aristotle differentiates *pre-existent knowledge* from *scientific knowledge*. Whereas the latter is produced by means of a logical proof, the former is produced by one's *intuition*. He goes on to argue that what we intuitively understand is indeed knowledge, in

the same way that what we prove is knowledge. The difference, he argues, is that axioms are *self-evident*. For example, the reflexive property of the natural numbers (i.e. for every natural x, $x = x$) is typically considered so obvious that proof is unnecessary. At the same time, the statement is so fundamental that it cannot really be proven. Despite their unprovability, Aristotle argues that axioms such as these constitute knowledge that is simply, unquestionably *known*:

> Our own doctrine is that not all knowledge is demonstrative: on the contrary, knowledge of the immediate premises is independent of demonstration. (The necessity of this is obvious; for since we must know the prior premises from which the demonstration is drawn, and since the regress must end in immediate truths, those truths must be indemonstrable.) Such, then, is our doctrine, and in addition we maintain that besides scientific knowledge there is its originative source which enables us to recognize the definitions.

Euclid apparently agreed with this epistemology because Book I of *Elements*, in addition to its definitions and propositions, contains a number of axioms, which are labeled either as *postulates* or *common notions*. In antiquity, a postulate denoted an axiom of a particular science, whereas a common notion denoted an axiom common to all sciences. Today, we do not think of axioms in this way. Rather, we simply consider them starting points for proofs. Euclid's axioms are given below:

Postulates

1. Hi

Common Notions

1. Hi

Euclid's proofs can also be described as *constructive*.

2.3. Ancient Knowledge, Lost and Found

2.4. Particles and Forces

2.5. Waves and Fields

2.6. Old Quantum Theory and Relativity

2.7. Quantum States and Operators

2.8. Information Theory

3. Information and Communication

For Wiener, entropy was a measure of disorder; for Shannon, of uncertainty. Fundamentally, as they were realizing, these were the same.

—James Gleick

3.1. Signals and Systems

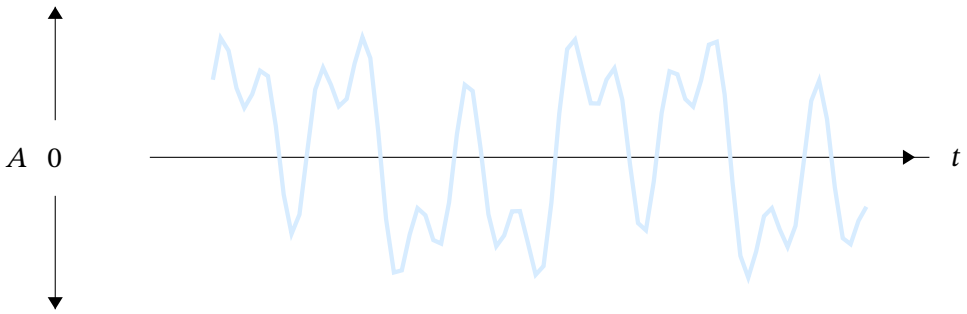
Data are the building blocks of *signals*, which are mathematical models of quantities that change over time. Signals are carriers of *information* about these quantities. A piece of information, known as a *message*, is encoded as data of a certain *medium*, either physical or numerical. In this sense, one can consider information a meaningful pattern that is embedded in a phenomenon. Philosophically, information is the *essence* of a *substance*, a set of properties that defines what a thing is. Mathematically, signals are functions f that map a set of times t to a set of amplitudes A, and they are classified according to the properties of these sets.

Sample rate is a property that is related to time. *Continuous* signals are functions of continuous time. That is, t is either the real number line $\mathbb{R}$ or an interval of it. In this case, t is also called a *linear continuum*. This definition of continuity is different from those given in real analysis, which describe functions for which sufficiently small changes in input result in arbitrarily small changes in output. For this reason, these signals are also called *continuous-time* in order to avoid any confusion with the continuous functions that are described in mathematics. If a signal is not continuous, it is *discrete* or discrete-time.

Continuous signals that have an infinite number of possible amplitude values (i.e. those that are modeled by $\mathbb{R}$ or $\mathbb{C}$) are called *analog* signals because an infinite-valued output can be viewed as *analogous* to the seemingly infinite complexity of real-world phenomena. At any instant in its time domain, an analog signal can *exactly* model a time-varying, real-world quantity, such as voltage, pressure, distance, mass, color, or anything else that is quantifiable. All continuous signals are analog, but not all analog signals are continuous. For example, the analog data of a continuous signal can be sampled at a finite rate in order to construct a discrete, analog signal.

In general, signals can be viewed as messages encoded in *data* whose decoded content is *information*. Like traditional written messages, ... Any signal can be expressed as an ordered sequence of data, each of which can be expressed as a tuple $(t, A(t))$. Because there are an infinite number of "instants" in any given interval of continuous time, the continuous signal shown above can be expressed as an *infinite*, ordered sequence of analog data.

Consider an arbitrary continuous signal:



Analog data are special because they exist in continuous space and thus theoretically have an infinite degree of precision. If we assume a classical model of physics in which time flows continuously, we can model physical quantities with real numbers by writing an equation $A = f(t)$ where $t, A \in \mathbb{R}$. In this sense, a complete set of analog data over a given time interval (i.e. an analog signal) can perfectly describe a physical quantity over the same interval. However, handling analog data can be very difficult.

Imagine that you throw a ball in the air and somehow exactly measure its height above the ground at every instant before it lands. If you were to organize these height data with respect to time, you would construct an analog signal that perfectly describes the height of the ball at any point during the throw. In practice, however, no measuring instrument has the required speed or precision necessary to do this. Additionally, you would need an infinite amount of memory to store continuous-time data that is collected at every instant.

The Rationale for Binary Numbers

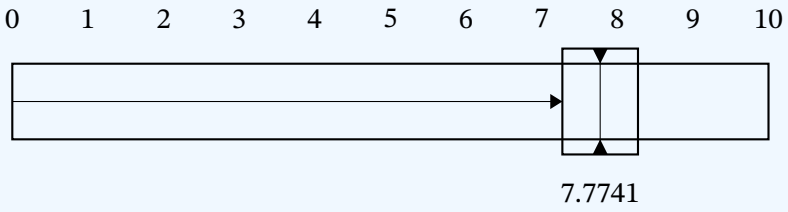
Binary encoding of data is not a requirement for computation. Rather, it is a concession that is made toward digital electronics. Computers have been designed for other bases, and, in some cases, a non-binary encoding is actually preferable. As long as the data can be encoded with a finite number of symbols, it can be handled by a *digital* computer (i.e. a computer that manipulates digits). A digital computer receives and operates on a *discrete* signal of data, a value or set of values that updates at regular time intervals (e.g. every second, every minute, etc.).

One could argue that *analog* computers process data that are encoded with an infinite number of symbols. However, this would require a loose definition of "symbol" because analog data cannot be perfectly represented by digits. Rather, analog data is represented in a real, physical medium, like position, pressure, or voltage. An analog computer receives and operates on this *continuous* signal of data that updates instantaneously with real-world time.

Consider the Mark 1A Fire Control Computer, a device, more appropriately termed a "calculator" by modern definitions, that was installed on United States battleships in World War II. This machine used mechanical parts, such as gears, cams, followers, differentials, and integrators, to perform the arithmetic and calculus necessary to hit an enemy ship with a projectile, even if both ships were moving in different directions. It also used smooth components that gradually translated rotational motion into linear motion along a continuum.

Despite being labeled with decimal (base-10) integers, this tool actually received

continuous, *analog* quantities, such as physical position, as input. If each of its mechanical positions were considered a different "symbol," this system would theoretically be able to represent uniquely every real number to a infinite degree of precision and, thus, compute continuous functions without error. Such analog computation is performed also in slide rules, as well as in early electronic computers, which used voltage to represent numbers instead of position.



Any digital encoding of numbers is possible in computing, and the choice does not affect what a computer theoretically can or cannot do. Rather, it has practical consequences. Decimal encoding was implemented in a number of early electronic and electro-mechanical computers, such as the Harvard Mark I and the ENIAC.

Regardless of which encoding is used, data is data.

Data are unorganized facts or statistical values that may reveal something of interest. They are collected by *sampling* the properties of *phenomena* from the observable Universe. Once collected, they can be cleaned, organized, and then analyzed in the hope of gaining a better understanding of reality. When a set of structured data is put into context, it can be interpreted as *information* that explains what has been observed. This information can then be *encoded* as an ordered sequence of data known as a *message*. Messages are *transmitted* and *received* both by Nature and by Man. In the latter case, this process can be viewed as the *communication* of abstract, human thought.

Data can describe a variety of things, such as quantities, qualities, and relations. In computer science, data are *quantitative*. That is, they describe *quantities*, which are properties that can be modeled with *numbers*. A quantity can be further classified as either a *multitude* or a *magnitude*. The former is a *discrete* quantity that is *counted*. One would ask "how many" elements belong to a multitude. In contrast, a magnitude is a *continuous* quantity that is *measured*. It is the size of something that is modeled as a *continuum* (e.g. by fields such as $\mathbb{R}$ or $\mathbb{C}$), and one would accordingly inquire as to "how much" of it there is.

A set of data collected at a finite rate (e.g. by a *measuring instrument*) is technically a multitude of samples, even if the quantity being measured is a magnitude. For example, if a ball were thrown in the air, one could ask "how much" height it has at regular intervals of time. Once the data collection ends, one can also ask "how many" samples were acquired. Furthermore, one can ask "how often" samples were taken. This is an evaluation of the *sampling rate*, and it is really just an alternative way of asking with "how much" speed samples were taken. Rates and ratios are still quantities. The existence of the rational numbers $\mathbb{Q}$ are evidence enough of this.

Data can be represented by various *media*, both symbolic and physical. In the age

of modern computing, data are often assumed to be *digital* in nature, expressed in the symbolic medium of numerical *digits*. This is the case for data processed by modern, electronic computers, which charge capacitors in order to generate voltages that represent *logic levels*. A *transistor* acts as a gate to each capacitor, staying closed to retain *electric charge* during storage and opening to measure or modify voltage on reads and writes respectively.

The ratio of a capacitor's charge to its capacitance determines its voltage (i.e. $V = \frac{q}{C}$). This voltage is then mapped to a logic level, which corresponds to a range of voltage values. These logic levels represent digits. In modern computer memory, data are represented as binary digits or *bits* (i.e. the digits 0 and 1), each of which represents a binary number. In this case, there are two logic levels, *low* and *high*.

Table 3.1.: Voltage Ranges for CMOS Logic Levels (V)

Signals	Low (0)	High (1)
Input	0.00–1.50	3.50–5.00
Output	0.00–0.05	4.95–5.00

However, quantitative data need not be expressed with symbolic digits. They can instead be defined in terms of an *analogous* physical quantity. For example, the unit of measurement known as the *millimeter of mercury* (mmHg) defines pressure in terms of the height of a column of mercury in a manometer. It is used to quantify blood pressure, and it does so by considering height an *analog* or *model* of pressure measuring that instead. Using this method, data about one phenomenon can be encoded as data of a different phenomenon, provided that the quantities involved are somehow related to each other. In this case, a height of 1 mm is related to a pressure of 133.322 Pa above atmospheric pressure.

3.2. Semiotics, Language, and Code

However, the term "universal language" is also often used in other contexts to describe a hypothetical language that all of the world speaks and understands. This is an old idea, and it is addressed in a number of myths and religious texts. In Genesis, the myth of the Tower of Babel explains the diversity of language as a result of God thwarting the Babylonians' plan to build a tower that would extend into the heavens. As punishment for their blasphemy, God "confused their tongues" and dispersed them across the world, shattering their once universal language. Similar myths are told by cultures across the world, such as the Native American Tohono O'odham people, who tell tales of Montezuma attempting to do the same, attracting the ire of the Great Spirit.

Alas, no such language exists and it is unlikely that one has ever existed. Rather, human language is generally believed to have evolved independently around the world from prelinguistic systems of communication such as gesturing, touching, and vocalization about 100,000 years ago. Similarly, written language evolved from proto-writing, the repeated use of symbols to identify objects or events.

Writing is distinct from speaking in that it is a reliable method of storing and transmitting information. Before the invention of written language, important pieces of history and literature were preserved through *oral tradition*, passing from one generation to the next through repetition and memorization. However, oral tradition is prone to data loss and unintended changes. Like messages in a game of telephone spanning centuries, stories were at risk of losing old details and gaining extraneous ones. Additionally, if a society fell apart, their stories could be lost forever. Writing is a tool that mitigates these issues by *encoding* speech into a symbolic code that can be inscribed onto durable media, such as clay tablets or stone, or onto more delicate, portable media such as parchment or paper.

3.2.1. Writing Systems

A *symbol* is a syntactic mark that is understood to represent a semantic idea. For example, the numeral "2" is a symbol for the abstract concept of the number known as "two," the quantity of items in a pair. Symbology is universal for mankind, as it is an exercise of our capacity for abstract thought. Humans can leverage the pattern-recognizing power of the brain to associate symbols or sequences of symbols (also known as *strings*) with ideas. Thus, we can proliferate information textually by means of a *writing system*.

…

An argument of any kind must be expressed in a *language*. In the case of an informal argument, such as a debate, a *natural language* is typically used (i.e. a language, spoken or written, that evolved *naturally* as an aspect of human culture). This is done in the name of *universality*. For example, a political debate that is held and perhaps broadcast in Poland would likely be conducted in Polish because Polish politics are primarily of interest to Polish people. Many Poles speak English in addition to Polish, but there is no language more widely spoken in Poland than Polish, with a whopping 97% of the country declaring it as their first language. For media directed toward the population of Poland, there is no language that will better ensure an effective dissemination of ideas. It is a nearly *universal* language in this context, a language understood by all intended recipients.

Formal proofs, on the other hand, require a *formal language* for clarity and precision. In practice, formal proofs are rarely written or read by logicians or mathematicians. Typically, mathematical proofs are written using a combination of formal and natural language. However, computer programs, written in *programming languages*, are considered formal.

A formal language is constructed from an *alphabet* Σ, which is a set of symbols. The set of all possible finite-length *words* or *formulas* that can be built from this alphabet is denoted Σ^*. A formal language L over an alphabet Σ is then defined as a subset of Σ^*. Thus, a language is a set of purely syntactic objects. It may conform to a *grammar* that specifies how words can be produced or arranged in a *sentence*, but a language is never inherently meaningful. The semantics of a language is always interpreted separately from the syntax.

3.2.2. Numeral Systems

4. Logic and Mathematics

> *Upon this first, and in one sense this sole, rule of reason, that in order to learn you must desire to learn, and in so desiring not be satisfied with what you already incline to think, there follows one corollary which itself deserves to be inscribed upon every wall of the city of philosophy:*
>
> *Do not block the way of inquiry.*
>
> —Charles Sanders Peirce

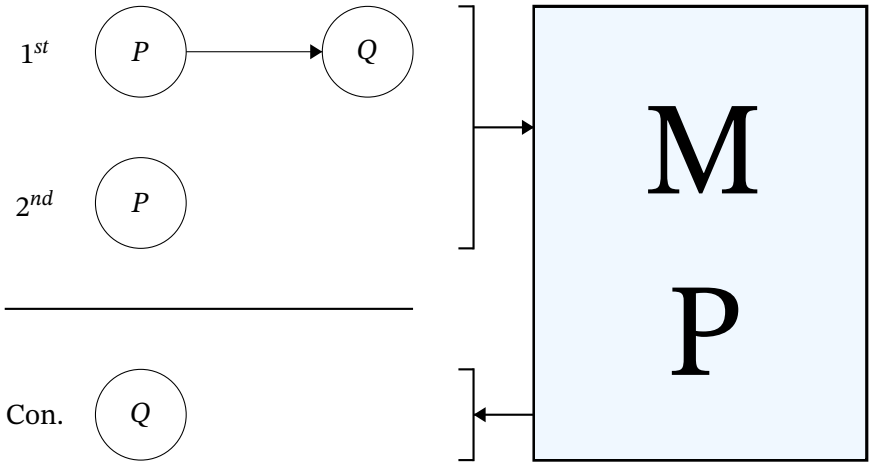
4.1. The Syntax and Semantics of Argument

Reasoning is a cognitive act performed by rational beings. It involves the absorption and synthesis of presently available information for the purpose of elucidating knowledge. It may require the ingenuity of thought, but in some cases the rote application of simple rules is sufficient. Reasoning could also be described as the providing of good *reasons* to explain or justify things. However, as you might expect, there is much debate over whether or not any particular reason is "good." Reasoning is broad, and it comes in different flavors, but ultimately it is the pursuit of *truth*.

Reasoning may involve the application of *logic*. This *logical reasoning* is the kind of reasoning that can be expressed in the form of an *argument*. However, this argument does not need to be "correct." Such reasoning simply must be explainable in *steps*, whether that be through informal speech or formal writing.

In logic, an argument has two parts: a set of two or more *premises* and a *conclusion*. If the conclusion *necessarily* follows from the premises, the argument is called *valid*. To know whether or not an argument is valid, one must produce a step-by-step *deduction* of the conclusion from its premises.

A deduction is constructed within a *deductive system*, which also has two parts: a set of two or more *premises* and a set of *rules of inference*. A rule of inference can be thought of as a kind of logical function. It takes statements (such as premises) as input and outputs either an intermediate result or the conclusion. A deduction, then, has *three* parts: a set of two or more *premises*, a *conclusion*, and a set of *intermediates*. A single-step deduction is represented below, along with a suitable rule of inference labeled MP. Note that the set of intermediates is empty in the case of deductions with only one step.



This is the quintessential rule of inference used in logical reasoning, expressed here in symbols. It is known as *modus ponens* (Latin for "a method that affirms"). The first premise $P \rightarrow Q$ is called a *conditional statement* where $\rightarrow$ is the *implies* operator. It states that if the *antecedent* P were true, it would *imply* that the *consequent* Q would also be true. The second premise P simply states that the statement P is indeed true. Thus, by means of *modus ponens*, we can *infer* from the premises $P \rightarrow Q$ and P that the conclusion Q must be true.

Inference is also called *entailment*. Though these terms describe the same logical concept, the latter more aptly describes the relationship between statements that are connected in a deduction. To infer is to come to a conclusion that is not *explicitly* stated in the premises. To entail is to require that something follows. If P *entails* Q, this suggests that Q is a *logical consequence* of P, which we can express using a *turnstile*: $P \vdash Q$. Inference is an action that humans perform. Entailment is a relationship between logical rules.

When the inferences made mirror the entailment, you've got yourself a *valid* argument. However, this does not necessarily mean that your argument is *sound*. *Validity* is a property of arguments with a conclusion that can be *deduced* from its premises according to rules of inference. However, validity says nothing about whether or not the premises have a *meaning* that accurately describes reality. *Soundness*, on the other hand, is a property of valid arguments whose premises are *known truths* or *axioms*. The deduction of a sound argument is called a *proof*.

Validity and soundness complicate entailment. Now, an argument can be evaluated either in terms of its *syntax* or in terms of its *semantics*. Syntax refers to the arrangement of symbolic *words* (or, more generally, *tokens*) in a body of text. Semantics, on the other hand, refers to the *meaning* that can be *interpreted* from the text. Both of these concepts are integral to human language and indeed to the communication of information in general.

Consider the following valid deduction:

"If it is raining, it is Tuesday."	$P \rightarrow Q$
"It is raining."	P
"Therefore, it is Tuesday."	$\therefore Q$

This conclusion is a *syntactic consequence* of the premises ($P \vdash Q$). That is, when the argument is evaluated simply as a collection of symbolic strings that are manipulated according to rules, P entails Q.

Syntactic consequence is sometimes expressed with the following statement: "If the premises are true, the conclusion must also be true." However, while this statement does hold for all valid arguments, text evaluated syntactically does not have any notion of truth. It is simply a sequence of tokens. Thus, it may be better to think of valid arguments as "games that are played according to the rules." For example, chess has no inherent meaning associated with it, but it still has rules, and valid games of chess are those that comply with those rules.

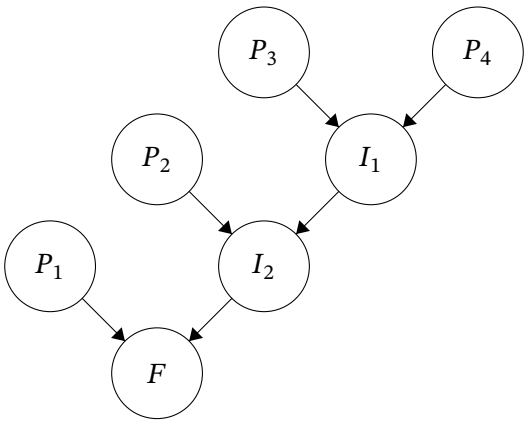
Now, it might be Tuesday and raining as you are reading this. However, semantically, it is not required to be Tuesday if it is raining. The first premise given above is, in reality, false. However, if we change it to something that we consider true, we can come to a conclusion that is both valid and sound:

"If it is raining, it is wet outside."	$P \to Q$
"It is raining."	P
"Therefore, it is wet outside."	$\therefore Q$

In this case, the conclusion is a *semantic consequence* of the premises ($P \vDash Q$). In addition to the argument being syntactically logical, it is also semantically true "in all universes." That is, there is no possible scenario where it is raining, and it is not wet outside. This syntactic-semantic difference will come up repeatedly throughout this guide.

4.2. The Structure of Proof

Of course, arguments can have more than one step. An argument can involve many inferences, some of which may use *intermediate conclusions* as premises for further conclusions. Multiple threads of reasoning can extend from the premises, weaving through intermediate conclusions and intertwining via inference until they all meet at a *final conclusion*. This is the *tree structure* that is inherent in a *formal proof*.



In this *tree*, the *first premises* are labeled as P_n where $n \in \{1, 2, 3, 4\}$, the *intermediate conclusions* are labeled as I_m where $m \in \{1, 2\}$, and the *final conclusion* is labeled F. Though one can make a distinction between first premises and intermediate conclusions, it is important to note that they are all premises with regard to the eventual final conclusion. Thus, there are no restrictions on combining them beyond those dictated by the rules of inference.

In modern proof theory, proofs like these are studied not only for their semantic meaning, but also for their mathematical structure. They are treated like *data structures*, which are abstract objects that are used in computer science to model the storage and flow of digital *information*. Similarly, a proof is an abstract text that models the flow of logical information as it is manipulated by rules of inference toward a final conclusion.

Information, in the context of computer science and, more generally, *information theory*, refers to a *syntactic* message. Formally, it is an order sequence of symbols

In computer science, the data structures of interest are *recursive*. That is, the structure can be defined "in terms of itself." *Recursion* is a deep topic, and we will spend much of the next part characterizing it, but a practical way to think about a recursive structure is in terms of *type*.

Types are a widespread feature in programming languages. As a concept, they are founded in a logical system called a *type theory*. There are multiple type theories, and some of them are used as alternative foundations for mathematics. We will cover all of this in detail later, but, essentially, a type is a label that can be given to

However, not all forms of reasoning have such strict rules. There are other methods of reasoning that cannot be modeled by an argument. In contrast to logical reasoning, *intuitive reasoning* has steps that are *not* understood. Although the question might seem peculiar, it is worth asking whether or not computation can be intuitive. So, to begin our journey of understanding computation in a modern, logical sense, we will first walk in the other direction, considering it instead in a mystical, otherworldly sense.

4.3. Intuitive Reasoning: The Oracle, the Seer, and the Sage

Intuition is the capacity to create conclusions without evidence, proof, or a combination of the two. If any premises are involved, they may appear to an onlooker as if they were plucked out of thin air. If any method is involved, it is esoteric or hidden from sight. Acts of intuition range from the mundane procedures we perform without thinking to great feats of intellectual, artistic, and athletic achievement. And while it is often associated with magic or supernatural ability, intuition is a real, observable phenomenon, and it is a form of reasoning.

Like that of reasoning, the definition of intuition is fuzzy. There are a variety of events that one might label as the product of intuition that are actually quite functionally distinct from one another. For this reason, I would like to consider and compare three archetypes that are known for their intuitive skills: the Oracle, the Seer, and the Sage. For the skeptics among you, I ask that you suspend your disbelief for a moment and assume that our characters are acting in good faith. There is no lying here; the conclusions are sincere.

The Oracle

An Oracle is a person who predicts the future by acting as a vessel for a god or a set of gods. They are considered by believers to be portals through which the divine speaks directly. They are found in histories all over the world, but most people associate the role with the priestesses of Ancient Greece. Picture a woman with eyes that roll back into her head as she speaks in a possessed, thundering voice.

For an Oracle, the conclusions come straight from the source. She may not even do any reasoning herself, save for the *unconscious reasoning* that is performed while she is possessed. However, conscious or not, she provides a great service to mankind. There is no clearer answer than the one given to you directly from the gods themselves, even if it has to be sent through what is essentially the human version of a telephone.

For those seeking counsel, the premises of the Oracle's conclusion are unnecessary because they just *know* that the statement must be true. For them, the connection the Oracle has with the gods and with reality is part of life itself. We all have deep beliefs like this. For example, most people do not feel that gravity needs to be proven to them in order for them to accept it as fact. Their perception of gravity in the world around them is proof enough. This is the kind of intuition characterized by *subconscious reasoning*, the reasoning you do without being aware of doing it.

The Seer

A Seer is a person who predicts the future by interpreting signs from a god or a set of gods. Unlike an Oracle, a Seer speaks divine truth in his own words, drawing from an innate, sometimes god-given power to see meaning in natural events or occult objects. There are various methods of divination that are used by Seers, such as scrying (the gazing into magical things, like crystal balls, for the purpose of seeing visions), auspicy (the interpretation of bird migration), or dowsing (the use of magic to find water, often with the aid of a dowsing rod). The acceptance of any particular method is cultural, but beliefs in divination vary greatly, even within a single society.

The Seer has premises for his conclusion, but the rules of inference involved in his reasoning are incomprehensible to others. He

The Sage

What is Truth?

Truth, in the absolute sense, has been discussed and debated since the dawn of man. It is a concept that seems obvious to us, and yet it always seems to elude our understanding. Philosophers have formulated dozens of theories of truth over the years, with some asserting that truth is an objective property of our universe and others asserting that truth is a useful lie that mankind has invented. And while there is merit in those claims that truth is not real, we still see around us the technology that was born out of our intuitive sense of true and false. Especially in computer science where everything is represented in binary.

The traditional theories of truth have been termed *substantive*. That is, they assert that truth has some basis in reality and that it is a meaningful thing to discuss.

Early theories of truth from Ancient Greece are considered the foundation of *correspondence theory*, the idea that statements are true if their symbols are arranged in such a way that they express an abstract thought that accurately describes reality. This theory defines truth in the context of a relationship between language and objective reality. It is a useful philosophy that allows us to orient ourselves in the world around us. However, many philosophers believe that truth cannot be explained by such a simple rule. In fact, there are many more factors that could play a role in the concept.

Objections to a strict correspondence theory usually take issue with the treatment of language as something monolithic and easy to classify within a true-false dichotomy. For example, a statement encoded as a sentence in a language is only meaningful to people who can read that language. What does this imply about a particular language's relationship with truth? Is a statement only true for those who understand it? Furthermore, there is no guarantee that readers of the same language will even be able to agree on the meaning expressed by a particular sentence. Languages often encompass multiple dialects that might parse words differently. Words themselves also may not be precise, and some abstract thoughts may not have suitable words in some languages. These are the points raised in *social constructivism*, a theory which avers that human knowledge is historically and culturally specific. Social constructivists also believe that truth is *constructed* and that language cannot faithfully represent any kind of objective reality.

Other substantive theories find the essence of truth nestled in other abstract concepts. *Consensus theory*, as the name implies, defines truth as something that is agreed upon either by all human beings or by a subset (such as academic groups or standards bodies). This is another anthropocentric definition that is at constant risk of philosophical division on any given topic. *Coherence theory* takes a more objective approach, claiming that a statement can only be true if it fits into a system of statements that support each other's truth. This is similar to the notion of *formal systems*, which we will discuss in depth later. However, traditional coherence theories attempt to explain all of reality within a single coherent system of truths, which is incompatible with our modern understanding of formal systems.

Modern developments in philosophy have resulted in theories that deviate from the long-held, substantive opinions on the nature of truth. These *minimalist* theories assert instead that truth is either not real or not a necessary property of language or logic. They claim that statements are *assertions* and thus are implicitly understood to be true. For example, it is understood that by putting forth the sentence "$2+2=4$," you are endorsing the semantic meaning "$2+2=4$ is true." The clause "is true" is called a *truth predicate*, and minimalist theories of truth often consider its use redundant.

This idea of a truth predicate is borrowed from Alfred Tarski's *semantic theory of truth*, which is a substantive theory that refines the correspondence concepts espoused by Socrates, Plato, and Aristotle for formal languages. This theory makes a distinction between a statement made in a formal language and a truth predicate, which evaluates the truth of the statement. Tarski made this distinction in order to circumvent the *liar's paradox*, which is often presented with the following example: "This sentence is false." If the predicate "is false" is considered to be

part of "this sentence," the truth of this statement cannot be decided. For this reason, Tarski states that a language cannot contain its own truth predicate. This is enforced by requiring that "this sentence" be written in an *object language* and that "is false" be written in a *metalanguage*. *Convention T*, the general form of this rule, can be expressed as

"P" is true if and only if P

where "*P*" is simply the metalanguage sentence *P* rendered in the object language. That is, the *syntactic representation* is assigned a *truth value* of true if and only if the semantic meaning it represents is considered true (according to whatever theory of truth you employ). By this rule, we say that "$2 + 2 = 4$" is true if and only if *the sum of the number* 2 *with the number* 2 *is equal to the number* 4. Minimalist redundancy theories modify Tarski's theory by interpreting "*P*" as an implicit assertion of the truth of *P*. The sentence "*P*," which asserts that *P* is true, is then false if and only if its semantic meaning *P* is false.

The debate on the nature of truth rages on *in perpetuum*. However, for our purposes, we must be practical. There is truth in logic and mathematics and computer science as well, but, in practice, it has little to do with the debates on *absolute truth* described above. Their truth is *relative*. That is, it relies on the assumption that our premises are true. Cognizant of this, we move forward with our thinking, searching for truths within these arbitrary boundaries.

4.4. Logical Reasoning: The Mathematician, the Scientist, and the Detective

Logical reasoning is the *inference* of new information from present information. It involves *rule of inferences* that are used to relate sets of *premises* to *conclusions*. There are three kinds of logical reasoning: *deductive*, *inductive*, and *abductive*. Each is classified according to which piece of information (premises, rule, or conclusion) is missing and must be inferred from the others.

Inductive arguments exist on a spectrum between *weak* and *strong*. Those that are stronger and more persuasive have a higher probability of having a true conclusion. Inductive reasoning is associated with *science* and *critical thinking* because it allows one to make generalizations about complex phenomena given limited evidence. Unlike deductive reasoning, it attempts to find new knowledge that is not simply contained within its premises.

Statements made by induction are bolstered with evidence whereas deductive statements are as true as their premises. This leaves inductive reasoning susceptible to *faulty generalizations* and *biased sampling*. Induction must also assume that future events

will occur exactly as they have in the past, which is not always the case. For example, a turkey that is fed every morning with the ring of a bell may infer by induction that bell → food. However, he will see the error in his reasoning when the farmer rings the bell on Thanksgiving Day and instead slits his throat.

Abductive Reasoning

Abductive reasoning is the inference of a *premise*, given a conclusion and a rule. It is investigative in nature. For example, given a conclusion "The grass is wet" and a rule "When it rains, the grass gets wet," we might determine that rain is the best explanation for the wetness of the grass. Thus, we abduce that "It might have rained."

Like induction, abduction can also produce a hypothesis. However, abduction does not seek a new relationship between two previously unconnected statements. Rather, it uses established relationships to find a reasonable explanation for a statement that is assumed to be true. It is often used by detectives or diagnosticians who need to find a probable cause of an event. It is also used in Bayesian statistics. While multiple premises may be abduced, typically we want to abduce a single, "best" premise.

Abductive reasoning allows us to ignore the many causes that are unlikely in favor of those few that may be relevant to the problem at hand. For example, doctors are often taught to heed the following proverb: "When you hear hoofbeats, think of horses, not zebras." That is, when a patient exhibits certain symptoms, a doctor should abduce from them a commonplace disease before considering more exotic possibilities. However, "zebras" do exist. Sometimes the most likely cause is not the actual cause. For this reason, abduction is also considered to be equivalent to a deductive fallacy called *affirming the consequent*, which is like a *modus ponens* performed in reverse. That is, given a conditional $P \to Q$ and the consequent Q, abduction infers P from Q by assuming that the converse $Q \to P$ of the conditional is also true. This is not a deductively valid inference. Considering our example again, the grass might be wet from rain, but this is not *necessarily* true. It is also possible that the sprinkler system is on. Or perhaps there is a zebra-esque scenario like a flood.

What is the strange loop? What is wisdom?

Part II.

Theory of Computation

> *Theory is relevant to you because it shows you a new, simpler, and more elegant side of computers, which we normally consider to be complicated machines. The best computer designs and applications are conceived with elegance in mind. A theoretical course can heighten your aesthetic sense and help you build more beautiful systems.*
>
> —Michael Sipser

The *theory of computation* is a field of study that is related to both mathematics and computer science. It is concerned with a process known as *computation*, which refers to any kind of *calculation* (the transformation of one or more inputs into one or more outputs) that is accomplished by following an *algorithm* (a set of rigorously defined steps). It is also concerned with the formalization of theoretical models of computation known as *abstract machines* or *abstract automata*. Also of interest are the *computational problems* that can be solved by these abstract 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.

Much of the work in the theory of computation is... quite abstract. While calculation and computation have been practiced by humans for thousands of years, the concept was only formalized by mathematicians, logicians, philosophers, linguists, and computer scientists in the past 150 years. This part of the guide will delve into some fairly academic topics with a decent amount of mathematical rigor, but it is written to be accessible and entertaining.

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.

5. Automata Theory

Automata theory is the study of *automata* and their behavior. 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, mechanical watches, robots, and computer processors as well as *abstract machines*. An abstract machine is any formal description of *mechanical behavior*. An abstract machine that is used to formally describe computation is called a *model of computation*. This is a mathematical description of what a computer is, and it models the computation that a real, concrete computer could actually perform, if you were to implement that computer according to the model's specifications. Like their concrete counterparts, abstract machines have *state*, a set of information in the *present* that was acquired due to *past* events. All automata begin with a *starting state*, receive some sort of *input*, and *transition* to another state based on the value of that input and the value of its current state.

One *abstract automaton* in particular fully models not only the computation that is performed by today's state-of-the-art computers, but also the very process of computation itself. We will spend a lot of time in this guide discussing the structure and behavior of this model as well as what its limitations imply regarding the future of computing, software, mathematics, and the meaning of life. We will also briefly explore the abstract automata that exist in pure mathematics and model computation that is not physically realizable.

5.1. From Knots to Nodes

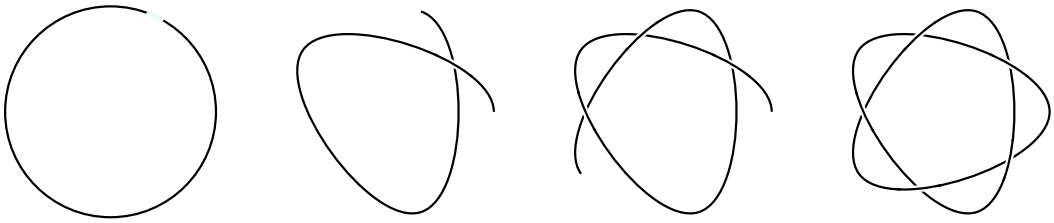
It is worth pausing here to think about what the word *node* really means. It comes from the Latin word *nodus*, which means *knot*. A node is like a knot in a string: a distinguishable lump at a certain position. And between any two adjacent knots in a string, there is a *link* ... of string. Similarly, between any two adjacent links, there is a knot.

The knots and the links cannot exist independent of each other. Without knots, there are no links of string—just one, continuous string. Without links, knots are totally unanchored and have no sense of position. Now, one could argue that a single knot is sufficient for the existence of links in a string, but such a view is naive and presumes a familiar, everyday sort of string.

For example, yes, you can tie a single knot in a shoelace, and you might then call the floppy bits on either side of it "links." But when you call something like this a link, what you are really pointing out is its *finite length*. That is, the "string link" *starts* at the knot and *ends* at the tip of the aglet. But length is a nonessential detail for links—the essence of a *link* is its *connecting* of things. A better term in this case would be *loose end* or simply *end*. A knot with two loose ends is shown below:

For our purposes, it is better to envision a taut string that stretches to infinity in both directions and to think of a *link* as a *connector of a knot pair* whose length may or may not be important. With this model, there must be at least two knots in the string if we are to think of them as *nodes* (i.e. things that are connected to something else). A string with five *trefoil* (three-leafed) knots and four links is shown below:

In the mathematical field of *knot theory*, knots are defined similarly, but they do not have ends. That is, a mathematical knot is defined as a *closed* circle embedded in a three-dimensional Euclidean space and subject to continuous deformation. Essentially, this means that you can cut a circular string at a point, tangle it, and close the circle at the same point to form any mathematical knot you like. The process of tying a *cinquefoil*, a knot with five *crossings*, is given below as an example:



Note that because mathematical knots are closed, they cannot be linked together like knots in a string. Instead, a link in knot theory is a collection of knots that are knotted together.

5.2. 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.

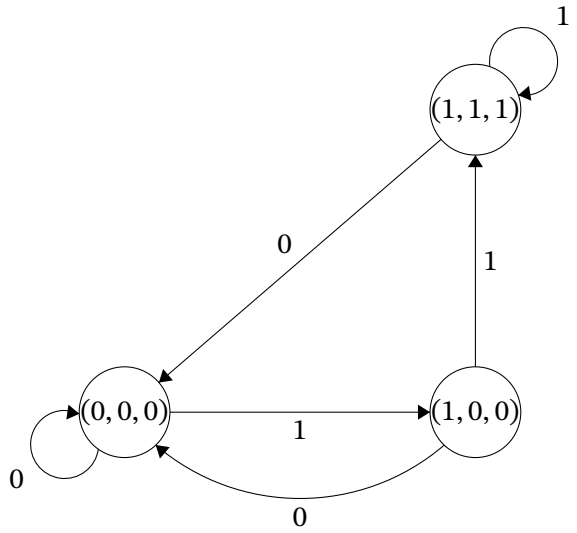
The machine's components here are the plank, the bird, and the music box. The components each have two states: the plank can be retracted or extended, the bird can be still or flapping its wings, and the music box can be paused or playing. The input to this machine is whether or not the clock hands both 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

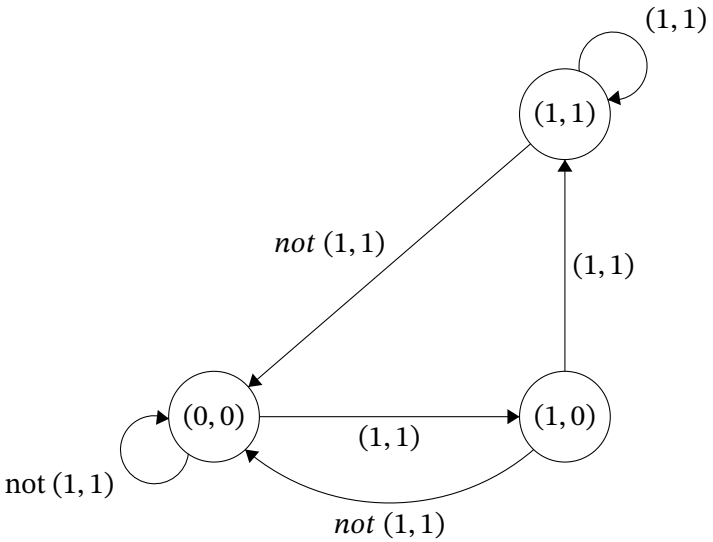
switches to 1, and the automaton transitions to the state (1,0,0). As the plank extends and transitions its own state from 0 to 1, the bird remains still, and the music box remains paused. Once the plank is fully extended, we are in state (1,0,0).

(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).



Note that the actual complexity of the concrete cuckoo clock would be a bit more complicated than the automaton 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 be modeled by a different automaton. To fully model a concrete automaton, we must reduce it to an abstract machine, a system of states and transitions that account for every intended behavior. Abstract machines are capable of modeling systems of any arbitrary complexity. The automaton above is a model of computation called a *finite-state machine* (FSM), and it will be the first abstract machine that we will discuss.

5.3. Formalization of an Abstract Machine

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 $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$.

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* $\to$ *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 $N = \{\text{SENTENCE, NOUN, ADJ}\}$, $\Sigma = \{\text{the, dog, building, plays, fluffy, good}\}$, and $S = \text{SENTENCE}$. The rules of P are given below.

- SENTENCE → the NOUN plays
- NOUN → ADJ NOUN
- NOUN → dog
- NOUN → building
- ADJ → fluffy
- ADJ → 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 → the NOUN plays → the ADJ NOUN plays → the good NOUN plays → the good dog plays
- SENTENCE → the NOUN plays → the ADJ NOUN plays → the ADJ ADJ NOUN plays → the ADJ ADJ ADJ NOUN plays → the fluffy ADJ ADJ NOUN plays → the fluffy good ADJ NOUN plays → the fluffy good fluffy NOUN plays → 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*.

5.4. 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*. Formal languages are similar to the *natural languages* that humans speak in that they have *words* that conform to a *syntax* that is specified in terms of symbols from an *alphabet*. These words can be arranged according to a *grammar* to form *sentences*, from which *semantic meaning* can be interpreted. However, unlike natural languages, they are defined in precise, unambiguous terms.

A formal language L over an *alphabet* Σ is a *set of words* that is a subset of Σ^*, the set of

all possible words that can be composed using Σ. Typically, formal languages conform to a *formal grammar*, which is a set of precise grammatical rules. However, not all formal grammars are equal. Grammars determine how words can be arranged into sentences, which determines the kinds of thoughts that can be expressed. Thus, grammar controls how *powerful* or *expressive* a language is. If we intend to use a particular formal language to instruct a computer to solve a problem, the *expressiveness* of the language will determine the kinds of 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.

In this section, we will examine automata with different *mechanical structures*. The *data* that are stored in components of these structures can be interpreted as words, and the structures themselves can be used to represent the rules of a formal grammar. However, some expressive grammars cannot be represented by simpler structures. Thus, the mechanical structure of an automaton acts as an *upper bound* on its *capacity* to understand language, and, by extension, its computational versatility.

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.

5.4.1. 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*, ∅, 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:

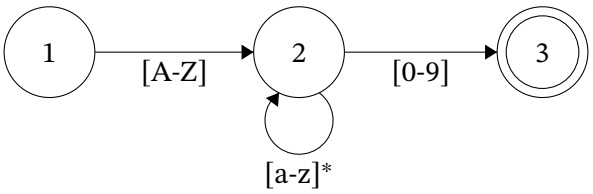
- The *concatenation* of A and B, AB, which denotes the set of strings that can be created by concatenating a string in A with a string in B.
- The *alternation* of A and B, AB, which denotes the union set of A and B.
- The *Kleene star* of A, A^*, which denotes the set of all strings that can be created 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.

5.4.2. 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 $A \rightarrow \alpha$ 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.

5.4.3. 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...

5.4.4. 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.

5.5. 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 likewise 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 5.1.: 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* 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. Modern computers can be described as random-access machines because they use *random-access memory* (RAM). In RAM, the time to access information is independent of physical location. From a performance perspective, this means that we do not have to consider *where* we store things in memory. It's all uniformly fast. This allows for the construction of *node-based* data structures, which we will discuss in a later section.

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 $\rho \in Acc$. 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*.

6. Computability Theory

> *Thirty years ago, we used to ask: Can a computer simulate all processes of logic? The answer was yes, but the question was surely wrong. We should have asked: Can logic simulate all sequences of cause and effect? And the answer would have been no.*
>
> —Gregory Bateson

Computation is mathematical in nature. Underneath its shiny UI and immersive applications, a computer is simply a machine that can be programmed to calculate numbers (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.

6.1. The Scope of 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 problem be solved *mathematically*? Furthermore, what does it mean for a problem to have a mathematical solution? It turns out that the answer lies in the difference between *reasoning* and *logic*.

6.1.1. Informal Logic

6.1.2. Formal Logic

Logic is a long-standing, far-reaching field, and its definition has changed over the ages, but it can be broadly described as the study of *argument*.

The practice and analysis of logic can be done in many different ways. For example, the early study of logic was mostly done in the context of *natural language* arguments made during oration.

Modern logic, however, is *formal*. That is, it observes the abstract *forms* of arguments instead of the arguments themselves.

Formal logic is the study of *inference* with regard to *formulæ*. Formulæ, Latin for "small forms" or "small rules," are finite sequences of symbols from an alphabet. They are purely *syntactic objects*, much like strings of text. Inference is the act of processing a formula A and deducing that a formula B is a *logical consequence* of A. This means that, given a set of *rules of inference*, the string A can be transformed into the string B by means of a finite series of applications of said rules.

and they are studied in within *formal systems*. These systems of abstract thoughts are used to infer the existence of formulæ by means of logical deduction starting from a given set of axioms. A formal system has:

1. A finite *alphabet* of *symbols*, from which a formula may be constructed
2. A formal *grammar*, to which a formula must conform if it is to be considered *well-formed* in the system
3. A set of initial formulæ, known as *axioms*, from which inferences can be made
4. A set of inference rules, known as a *logical calculus*, which prescribe how inferences are to be made

Formulæ, which are finite sequences of symbols from a system's alphabet, are purely *syntactic* objects, but, given an *interpretation*, they can be endowed with *semantic* meaning. For example, an interpretation of first-order logic requires the following assignments of semantic meaning:

1. Variables are assigned *objects* (things that can be modeled) from a *domain of discourse* (a well-defined set of objects)
2. Predicates are assigned *properties* of objects
3. Formulæ, which contain variables and predicates, are assigned a *truth value* of true or false

A well-formed formula with such an interpretation is called a *sentence*, and the meaning expressed by a sentence is called a *statement* or *proposition*. A sentence expressing a true statement (according to the axioms and inference rules of the formal system) is called a *theorem*, and a set of theorems is called a *theory*. For example, ZFC set theory is a first-order *theory* of sets. It has a collection of theorems written in first-order logic that form an *axiomatic set*, and these theorems are used to prove other theorems that also belong to the theory using a set of inference rules.

6.1.3. 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.

6.1.4. "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 19^{th} 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.

6.2. 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."

6.2.1. 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 $\{1, 2, 3, \dots, \infty\}$. 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 $a_0, \dots, a_n \in \mathbb{Q}$, 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, ... 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!"

6.2.2. 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.

6.2.3. 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 x, $\mathcal{P}(S)$ has cardinality 2^x, 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 either true or false. According to this philosophy, 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 ($\mathbb{A}_\mathbb{R}^c$, 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 (א, 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

6.2.4. 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

6.2.5. 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.

6.2.6. 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.

6.2.7. 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

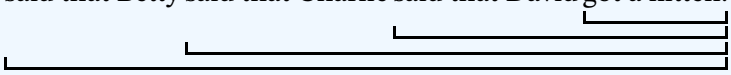
$$f(w) = \begin{cases} 1 & \text{if } w \in L \\ \text{undefined} & \text{otherwise} \end{cases}$$

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.

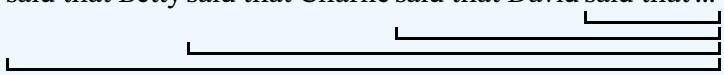
The following sentence is recursive (and, thus, also recursively enumerable):



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.

6.2.8. 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.

6.3. Hilbert's Program and Gödel's Refutation of It

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* questions. 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.

6.3.1. Hilbert's Second Problem

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. It had been assumed until this point that mathematics was something that could be trusted, that a statement could be proven true from previous mathematical truths. The creation of

Zermelo-Fraenkel set theory in the late 1920s resolved some of these paradoxes, such as Russell's paradox, but not all inconsistencies were resolved. Mathematics was still curiously "broken" in certain ways.

Metamathematics

TEXT

As a proponent of the philosophy of mathematics known as *formalism*, Hilbert believed that mathematics was not a description of some abstract part of reality, but was rather akin to a game where pieces could be moved according to a set of rules. He believed that a mathematical "game" was played with "pieces" from an arbitrary set of *a priori* truths. These pieces are called *axioms*, and they are manipulated according to an arbitrary set of *inference rules* to construct *conclusions*. These conclusions can then be used as *premises* for the inference of further conclusions. Thus, formalism avers that mathematics is purely *syntactic*. It is nothing more than the manipulation of strings of symbols by logical rules, and any sort of *semantic meaning* derived from these strings is merely the product of an *interpretation*.

In the early 20th century, Hilbert made it his life's goal to construct a set of axioms, from which all existing and future mathematical theories could be derived in a consistent manner. His peers 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 basic 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 deciding Cantor's continuum hypothesis.

Stated more generally, Hilbert intended to construct a *formal system* that was *complete*, *consistent*,

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.

6.3.2. Gödel's Completeness Theorem

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*.

All of Gödel's theorems that we will discuss in this section are statements about theories, which are represented as mathematical objects so that they can be studied mathematically. A theory is similar to a formal system, but it does not necessarily have inference rules. However, it *is* a set of axioms written in a language, which has an alphabet and a grammar. A theory can be coupled with a set of inference rules and often these rules are understood from context (e.g. by examining the logical structure of the language that the axioms are written in). Mathematical theories, for example, typically employ a first-order logical calculus because first-order logic is the standard formalization of mathematics. However, theories need not be mathematical. Political theories are still theories. They are composed of theorems expressed with the alphabet and grammar of a natural language. They have sets of fundamental beliefs from which other beliefs are derived, and they may be coupled with some form of deductive reasoning, though this reasoning may not necessarily be first-order logical. That said, we will focus on first-order logical theories written in the syntax of mathematics.

A theory coupled with a deductive logical calculus is called a *deductive system*. A formula that is found to be true, independent of interpretation, by means of this system (i.e. via a series of inferences that starts at the axioms) is called *logically valid*. The axioms of a theory are assumed to be true, so a logically valid formula deduced within a *truth-preserving* system must also be true, but only within the theory. All truth is relative, and, as such, the truth value of any given formula is necessarily dependent on the choice of axioms and the inference rules that are allowed. There is more than one way to reason about a given theory, and, as such, a theory can be coupled with different logical calculi to form different deductive systems. Examples of first-order logical deductive systems include *natural deduction*, the *Hilbert-Ackermann system*, and *sequent calculus*.

A theory is syntactic in nature. It is a set of strings written in a language $\mathcal{L}$, and these strings are called $\mathcal{L}$-formulæ. Any logical calculus that is applied to a theory $\mathcal{T}$ will simply map these strings to other strings according to some set of rules. Theories have no inherent meaning. On the other hand, a *model* is semantic in nature. A model $\mathcal{M}$ is an ordered pair $(\mathcal{T}, I)$ where $\mathcal{T}$ is a theory (a set of $\mathcal{L}$-formulæ) and I is an *interpretation function* with domain the set of all *constant*, *function*, and *relation* symbols of $\mathcal{L}$ such that

1. If c is a *constant symbol*, then $I(c)$ is a *constant* (a statement) in $\mathcal{T}$
2. If F is a *function symbol*, then $I(F)$ is a *function* on the domain $\mathcal{T}$
3. If R is a *relation symbol*, then $I(R)$ is a *relation* on the domain $\mathcal{T}$

6.3.3. Gödel's Incompleteness Theorems

6.4. The Entscheidungsproblem and the Church-Turing Thesis

6.4.1. μ-recursive Functions

6.4.2. The Untyped λ-calculus

6.4.3. The Halting Problem

Computability is Recursion

TEXT

6.5. Turing Degrees

7. Computational Complexity Theory

7.1. Big O Notation

7.2. Complexity Classes

8. Computer Architecture

8.1. Mechanical Computation

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.

Modern Computing is American

TEXT

Part III.

Types, Structures, and Algorithms

9. Type Theory

9.1. The Curry-Howard-Lambek Isomorphism

10. Algebra and Category Theory

Yay, abstract algebra! A theory of objects!

11. 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 11.1.: 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.

11.1. 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).

11.1.1. 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 $i \times$`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 11.2.: 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.

11.1.2. 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 11.3.: 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** = $O(1)$

11.1.3. Skip Lists

11.2. 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 11.4.: 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

11.3. 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 11.5.: 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

11.4. Deques

11.5. Priority Queues

11.6. 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 11.6.: 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 11.7.: 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: $O(V^2)$		

11.7. 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 11.8.: 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.

11.7.1. 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 11.9.: 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.

11.7.2. 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 11.10.: 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 11.11.: 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"

11.7.3. 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 11.12.: 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.

11.8. 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.

11.8.1. 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 11.13.: 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.

11.8.2. 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 11.14.: 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

11.8.3. 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 $O(logn)$ 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`.

11.8.4. Multisets (or Bags)

12. Algorithms

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

Brute-force description

Greedy description

Divide and Conquer description

Dynamic Programming description

12.1. 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.

12.1.1. 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 12.1.: 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.

12.1.2. 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 12.2.: 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.

12.1.3. 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.

Part IV.

Computer Programming and Operation

13. Programming Language Theory

13.1. Elements of Programming Languages

13.1.1. Syntax

13.1.2. Type Systems

13.1.3. Control Structures

13.1.4. Libraries

13.1.5. Exceptions

13.1.6. Comments

13.2. Program Execution

13.3. History of Programming Languages

13.4. Programming Paradigms

13.4.1. Imperative versus Declarative

Functional Programming

Logic Programming

13.4.2. Procedural versus Object-Oriented

13.5. Programming Techniques

13.5.1. Higher-Order Programming

Lambda Expressions

13.5.2. Currying

13.5.3. Metaprogramming

14. Specification and Implementation

If you have built castles in the air, your work need not be lost; that is where they should be. Now put foundations under them.

—Henry David Thoreau

14.1. Java

14.1.1. Java Standard Library

java.lang

java.util

java.io

java.net

14.1.2. Java Techniques

Static Initialization Blocks

Lambda Expressions

14.2. Haskell

14.3. Frameworks

15. Operating Systems

16. Practical Computing

"Stupid Computer Shit"

16.1. Linux

16.1.1. File Systems

16.1.2. Common Commands and Tasks

16.1.3. Userspace

16.2. Everyday Tools

16.2.1. Essential Programs

16.2.2. Version Control

Git

16.2.3. Unit Testing

JUnit

16.2.4. Build Automation

Make

GNU Make compiles source files into executables.

Maven

16.2.5. Virtualization and Containerization

Docker

16.3. Languages and Language-likes

16.3.1. Markup and Style

TeX

HMTL

CSS

16.3.2. Data Formats

XML

JSON

16.3.3. Data Query and Manipulation

SQL

Part V.

Software Craftsmanship

A delayed game is eventually good, but a rushed game is forever bad.

—Shigeru Miyamoto

17. Software Engineering Processes

18. Design Patterns

Part VI.

Appendices

A. System Design

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.

B. Lessons Learned

The fundamental cause of the trouble is that in the modern world the stupid are cocksure while the intelligent are full of doubt.

—Bertrand Russell

C. Writing Well

D. How *The Internet* Changed Humanity

E. Thoughts on Education

Most thought-provoking in our thought-provoking time is that we are still not thinking.

—Martin Heidegger

F. Passion

G. Bibliography

The origin of concepts, even for a scholar, is very difficult to trace. For a non-scholar such as me, it is easier. But less accurate.

—Peter Freyd

H. Glossary

Good words go here.

I. Future Work

The
End

Notes for the Introduction to Volume I: Philosophy of Computation

Data, Information, Knowledge
It's hard to fully understand these terms because they are so general.
But, because they are so general, they are applicable to a lot of different situations, so they are useful to understand.
They can model anything? Does that even make sense?
We studied and used these concepts long before computers.
In this chapter, we talk about the histories of physics, language, and logic because that is necessary for some reason.

In some sense, wisdom is obviously outside of the scope of computer science and software engineering. That is, it is not a part of their subject matter. It is not in any relevant textbooks or journals, and its study is not the direct focus of any expert working in those fields. But in another sense, wisdom is actually the ultimate goal of \\

In some sense, wisdom is obviously outside of the scope of computer science. In another sense, it is its ultimate purpose.
How is wisdom relevant to CS? Because wisdom is the ultimate purpose of life, and CS is a part of life.

Wisdom is the use of knowledge in the right way.
What is right is plainly before you. No matter how you try to obfuscate or twist it, the right thing to do stares back at you.
I associate wisdom with intuition.

Descartes, Meditations - he thought that metaphysical wisdom could be found by first doubting every thing that you know.
You may be surprised by the fundamental things that you take for granted, by how much of your misery and joy rests on pillars of sand. We cannot build an intuitive understanding on these hazy foundations.

Intuition allows you to not only solve problems but to see through problems. Seven Bridges of Konigsberg.

We must then be like babies. Cognitive bias.
Let us take a page from Hesiod and begin with the chaos of uncertainty. And let us also be like Descartes, building up from nothing.

Alternate Intro

GP1: True names. Identity. Categorization.

One of the longest and oldest threads that can be found in the tapestry of human culture is the concept of the \textit{true name}. It is mystical in origin, appearing in magical and religious contexts as well as in folklores throughout history, and it is founded on the belief that certain words are imbued with the power to change reality. A true name is an expression of its bearer's true nature or \textit{essence}, a summary of all that they are, and so it is said that to know such a name is to have power over the person or thing to whom or which it refers. Like an arcane spell in a dusty grimoire, it is a precious and potent bit of information in the eyes of those who believe. \\

Of course, magic is not real. That is, magic does not exist in reality, magic is not a special phenomenon that science cannot describe, and magical incantations do not grant casters supernatural abilities. But the curious thing about magic is that its existence depends on belief, and it is, in fact, really and truly real for those who think it is (or it is, at least, as really real as "real" gets). And, to make it all even more complicated, depending on how you define the word 'magic,' the slippery thing has a way of wriggling itself into the rigorous and brow-furrowed discussions of non-believers who dig too deep.

The true name is a supernatural tool for nullifying supernatural threats.

Magic might be real... depending on how you define magic.
Magic and feelings. Axioms, starting points, non-rigorous ideas, pure meaning.
"Incantation" as slang for a computer command

GP2: Computer science is not about computers. Or, at least, it is not *necessarily* about computers.

GP3: Computer science is actually about computation. Ok, then what is computation? A process or event. Pops up in many fields of study (logic, mathematics, linguistics, engineering) and was eventually studied itself (making computer science a very interdisciplinary subject). Performed for the purpose of resolving uncertainty.

GP4: Confusion is our natural state. By means of our intelligence, we give order to our surroundings.

GP5: Describe data, information, and knowledge with the help of graphics.

GP6: Domain / Theory / Model

GP7: Computation can be modeled like a game. There is a game world, with game pieces of varing properties, with various relations among them. An agent is a game piece with agency (i.e. the ability to execute functions).

Notes for Chapter 1: Cognitive Ontology

What is a ... number, set, number set, quantity, space, map, morphism, relation, function, field, algebra, operation, algebraic operation, pair, tuple, combination, permutation, vector, matrix, tensor, shape, structure, change (w/ qualifiers like linear, quadratic, cubic, exponential, logarithmic), order, metric, measure, geometry, topology, infinity, continuity, graph of a function, etc.
What is a ... concept, idea, notion, object, property, attribute, event?

A type specifies how information should be interpreted. With the correct interpretation, you acquire knowledge.

Upper Ontology

It is very difficult to build a linear progression of technical terms as fundamental as those found in this section. But we have to start somewhere. It would be helpful to have a root category: thing.

Dichotomy of Thing: every thing is either an object or a relation. And every object is either an individual, an attribute (or property), or a class (or set).
Upper ontology of Thing: Individual, Attribute, Class, Relation (and Function)
Relationship: <Thing> <Relation> <Thing>.
Structure
Ontology

Everything
Universe
Domain of discourse
Context/POV
Universal quantifier

Physical (system)
Perceptual (theory)
Conceptual (model)

Descartes

Locke (tabula rasa)
Hume (problem of induction)
Kant (transcendental aesthetic, noumena, phenomena)
Transcendental idealism is about finding which a priori concepts are necessary for conscious experience to be possible.
"Transcendental ideality of space"
The noumenal world may be a hyperspace. Many theories in modern physics rely on this assumption.
According to Kant, creating knowledge requires both sensibiity and understanding.
We are pre-programmed: space, time, causality, numbers, reflexes, sociability

Space is an intuition of outer-sense, time is an intuition of inner-sense.
Conscious experience requires an understanding of the self as a subject, an observer of objects. The presence of objects is required (i.e. you need to perceive something in order to separate one moment from the next).

SEP: Mereology
Model of hierarchical complexity
Double-loop learning

"A set is a plurality thought of as a unit."

Gestalt psychology, principles of grouping
We perceive spatially local, causally efficacious \textit{bodies} that persist in time to be physical objects. But an object can be anything that can be "thrown against" your perception.

Bootstrapping
Analog magnitude representation (cardinality)
Parallel individuation (up to 3)
Serial ordering (ordinality)
=> Children induce the successor function

Infinite regress
Primitive notions
Axioms restrict relations between primitive notions (they are constraints).

Physical -> Perceptual

Perception
Dichotomy (partition, division, individuation, differentiation, cut): pt()
Analytical thought
Self/other

Boundary
Separating hyperplane
Manifold of sensory data

Something
Parthood (Mereology, Mereonomy)
Reflexivity
Transitivity (parthood is NOT transitive, unlike set membership)
Antisymmetry
Existential quantifier

Duality (blue box on natural dualities: light/dark, good/bad, high/low, hot/cold, alive/dead)
Negation
Nothing. Can there ever be nothing? Or is nothing just an abstract idea?
Negated existential quantifier

Attributes (properties)
Quality vs. Quantity
Magnitude vs. Multitude

Attention
External perceptions
Internal perceptions

Text (or copy) is a perceptual thing. It is not the same thing as words on paper. It is abstract, but not as abstract as the concepts it represents.

Perceptual -> Conceptual

Conception (involves comparison of, reflection on, and abstraction from a set of percepts
Wiki: Concept
Holistic thinking

Collection (unification, integration): ref(...) or {...}
Classes (sets)
Extension (natural kinds)
Intension (types)
An idea is well-founded if it has a defined extension or intension. If your terms are vague, everything you say will be fluff.
We only have so much conceptual precision. We can't keep zooming in. Thus, we arrive at some most fundamental thing: the atom (or primitive (which really stands for "object of a primitive type/kind")). Every thing is made up of other things, except for atoms.
Similarly, we reach a limit in the other direction. Universes are things that cannot be parts of themselves

. They are just themselves.
Assimilation and Accommodation

Conceptual categories are "structures of the mind."
  Percepts are synthesized into a concept that can be categorized.
Try to keep in mind the definitions of "type" and "kind" in your use of everyday language. The quality of language itself changes entirely when one considers every word he or she chooses to say.

Conceptual -> Perceptual

Naming (inverse: recognizing, understanding)
Map-territory relation
Type-token relation (instantiation)
Name could refer to a concrete or abstract concept
Encoding (concept to percept)
Compression (reduce dimensionality, percept to concept)

Reference
Referential universe is isomorphic (structure-preserving) to referent universe
Extrinsic iso: both universes, part of nothing else
Intrinsic iso: both atoms, partless

Perceptual -> Physical

Communication

Order
Hierarchy (mathematically, a set with a preorder defined on it; a proset)
Taxonomy (nested or inclusion hierarchy)
Containment hierarchy (a taxonomy of strict sets)
Subsumptive containment hierarchy (subset relation, is-a, OOP class inheritance, general-specific spectrum)
Compositional containment hierarchy (mereonomy)

Dimension
Nominal (unordered, set-like)
Order (hierarchy)
Referential order
Conceptual order
Ontological priority
Causality (upward and downward causation)
Causation is a succession of events that is necessary and irreversible
Metric
Atom
Trivial dimensions
Space and time

Modality
Dimension represented with neural interconnectedness?
Curse of Dimensionality

Identity: spatial, temporal, referential, isomorphic
Analogical reasoning => isomorphism

Closed world
Open world

Contiguous/discontiguous
Continuous/discrete

Sentence
Noun
Proper noun
Count noun
Mass noun
Adjective
Adj order: opinion-size-age-shape-color-origin-material-purpose
Verb, adverb
Transitive
Intransitive
Gerund
Binary tree form: noun phrase + verb phrase
Dynamically-constructed concept
Ternary form: concept, copula, definition
Verb tense
Verb transitivity affects dimensionality

ADT -> DS -> Instance -> Output -> Thoughts
Con -> Per -> Phy -> Vir -> Con

Mereology vs. Set Theory
Function composition vs. set intersection
Functional vs. OOP?

Higher-order concepts
Metacognition
Turtles all the way down, concepts all the way up

The Hard Problem of Consciousness

Physicalists take one more step and say that the neutral elements are physical. But then there are those pesky qualia to consider. Can they be reduced to physical states? Reductive and non-reductive physicalism (or materialism).

Okay, we follow non-reductive physicalism to accommodate a mental conception of qualia. Will we ever understand

mental states in the way that we understand physical states?
Problems versus mysteries
Temporary mysterianism (maybe, but it will require a paradigm shift in science)
Permanent mysterianism (no, we are cognitively closed)
Colin McGinn - there are four approaches to explaining something (reduce it, make it fundamental, claim it is magical, eliminate it from reality)

Eliminative materialism - the antithesis of property dualism
Dennett: Wittgenstein, anti-theist, verificationist, critical of folk psychology and the intuitive understanding of one's own consciousness

Quining Qualia (1988) and Consciousness Explained (1991) - Daniel Dennett
Qualia are... ineffable, intrinsic, immediate, and private
This description is unscientific, so qualia do not exist in this sense
Objecting to the intuitive understanding of consciousness. This is a vital part of his philosophy, that people do not understand their own consciousness as much as they might believe they do. I agree with this.
Most philosophers of mind are atheists, but Dennett is an anti-theist. You can see it in his argument a bit. He seem to relish the idea of obliterating the last thing that made mankind special: the Cartesian mind. He also dislikes intuitive ideas, like folk psychology.
Multiple realizability

Which one takes precedence: what we observe or what we can verify?
For Nagel, the axiom is that "subjective consciousnness is real" because we constantly observe it. For Dennett, the axiom is "only objectively verifiable things are real" because science presupposes objectivity.
Dennett is a functionalist. Functionalism requires only access consciousness. He doesn't deny that we experience a phenomenal consciousness or a "subjective character of experience," but he thinks that those private, personal experiences, in and of themselves, have no place in reality. There is neurological activity that causes us to experience those qualia, but the experiences themselves have no quiddity, no place in the ontology of the actual world.

Mary's room - Frank Jackson's epiphenomenalist argument (1982)
USE THIS TO CONTRAST NAGEL AND DENNETT
Defines qualia

Ontological vs. epistemic readings
Epiphenomenalism - mental states are caused by physical states, but have no causal effects on the physical world.
Jackson later becomes a physicalist
Cover weak reductionist, strong reductionist, and eliminative materialist positions on the problem

Facing Up to the Problem of Consciousness (1995) - David Chalmers
Qualia are not functionally analyzable (inefficacious, purposeless)
Panpsychism
Quantum Mechanical Dualism
The hard problem of conciousness is a subproblem of the mind-body problem, which also takes thoughts into account.

Integrated Information Theory (2014)
Inspired by Chalmers' panpsychism, assumes that consciousness (phi) is actually a fundamental physical property of feedback systems.

What kind of things are qualia? Internal and external sensations, emotions, and moods. Felt states.

Functionalism, multiple realizability (qualia are defined by the function they serve, they are realizable by means of multiple implementations, not just one). Animals can feel the same qualia, even though their neural architecture differs. Artificial minds can experience qualia.

Why is all this qualia stuff important for CS?
There are multiple universes we could consider: physical (objective, subjective (perceptual, conceptual)). In CS, we work in a conceptual universe.
Hofstadter, strange loop
Formal systems
Hardware and software
Cognitive biases. Computers do not have them but programmers do, and computers only do what programmers tell them to do.
Feedback systems in IIT have nonzero levels of consciousness. Consciousness is not a desirable feature of tools, so it may be worth designing feed-forward systems instead to ensure that we don't accidently create conscious programs.

We are not concerned with subjective phenomena (experience). We are only interested in sensory data that is organized into information and integrated as knowledge

(cognition).
That is, we are not interested in qualitative notions. Computer science is an objective discipline. Assuming that the processor operates as instructed, it is actually a deterministic discipline (outside of randomized algorithms). Thus, we are only interested in quantitative notions.

The Categories of the Mind

Babies:
They are most interested in things that differ moderately from their schemata.
Assimilation and accommodation
Schemata, habits, pattern recognition (memory) based on pleasure and pain at 1-4 months
Intentionality at 4-8 months
Logical thought at 8-12 months

Language: nouns (things and concepts) and verbs (actions (active) and events (passive))

Patterns, mathematics is the study of patterns

Infants understand objects (continuous, spatiotemporal boundaries) before they understand properties (differing features).

Categorical perception
Categorical proposition
What is a category? Philosophy and math.
In classical philosophy, it is a predicate or highest kind.
In contemporary philosophy, it is a class of things that share a predicate (or many predicates). It is a bundle of properties.
In math, it is a class of objects and a class of morphisms. An object is a set perhaps with structure (properties). Morphisms are relations.
What is a category of being? An ontological category, a category for things that exist.

Important: is there a most general category? If there were, everything would belong to it, and it would provide no info. Thus, we need to differentiate. This is what a category system is: an interpretation of reality, a firm partitioning of some phenomena from others.

Phenomena and noumena.
Thing-in-itself

Categorizing in other fields: history periods, biology

taxa

An object is any thing that is suitable for thought (" thing presented to the mind"). Are all things objects? Who the fuck knows. Let's assume yes.

---OBJECTS------------------------------

Discuss how object and subject switched.

What is a metaphysical object? It's like a trope. It's a pattern that is recognizable, and it comes up frequently enough that there is a name for it. It's a description of some thing that often appears in reality. It is a \textit{type} of thing. We will discuss many of these metaphysical objects in the chapter Types and Structures.

Objects vs. events. Is there a difference? Maybe not, but there at least appears to be one. Humans have traditionally discerned a difference, notably in language. Perhaps the difference is only conceptually convenient. Is an apple distinct from its decaying over time? Is computer state really a meaningful object with clock speeds in the gigahertz? Is a thunderstorm an event, or just a system of many, tiny objects with different properties, all changing with time?

"Abstraction takes us further from reality but casts a wider light on it."
Abstraction allows you to handle large quantities of information as single units. This allows you to manipulate information more efficiently (say more, do more).

Are there abstract things at all (nominalism)? Is chess an abstract object? Is a novel an abstract object?
Concrete objects have substance. An object of spacetime is an event.
Is a set of books abstract or concrete?

---PROPERTIES---------------------------

Universal, particular, singular

Bundle theory, bare particularism.
has-a relation

The word data has heretofore been used in this text to refer to a multitude. In recent years, however, the

word has also been used as if it described a magnitude.
Everything digital is necessarily a multitude of bits.
However, these multitudes can be interpreted as discrete magnitudes.
** Old paragraphs on magnitude/multitude/data down in signals. **

All quantities are also qualities. It would be hard to argue that all qualities are also quantities in any meaningful way. For example, using millihelens as a quantitative unit of beauty. It doesn't really work.
optimal synonym

---CLASSES-----------------------------

Extension-intension (extensional and intensional definitions of class)
Individuals (urelements) and sets (individuals are deemed redundant in modern set theory). Pure and impure sets.
Subset, superset, class.
A natural kind is a collection of tokens that have something in common.

---RELATIONS---------------------------

Properties could be considered relational in nature.
Something that we would call hard is really just harder than other things. There is no absolute hard.

Identity and Sameness
Identity is tricky in philosophy but easy in math.
Sorites paradox, Theseus paradox
Indiscernibility of identicals

Symbols represent objects, but they are not the objects.
Models represent reality, but they are not reality.
Exoteric and esoteric.

The syntax, semantics, and pragmatics of a language (relating symbols to concepts, and doing so sensitive to context)
On Exactitude in Science
Simulacra and Simulation
The desert of the real

Syntax includes grammar, phonology, and orthography.
Truth-falsity.
Programming is the conversion of natural language

semantics to formal semantics written in formal syntax.
A high-level language just has a formal syntax that is closer to natural syntax.

--- ONTOLOGIES -------------------------

Maybe use abstraction/specification as your terms instead because abstraction is a mental process as well (link human behavior with logical structure).
Abstraction/specification is parallel to deduction/induction.
Abstract/concrete is a spectrum. An abstraction is a compression of subclasses into a single class. It is the identification of similarities between objects and the treatment of them collectively as an (abstract) object.
Abstract really means "not in space and time."
Hypernym, hyponym
Intuition is an abstract understanding that does not require conscious thought. It is characterized by generality. That is, it is applicable to many situations (without thought).

Subsumption - taxonomies
Instantiation
Aggregation, composition, containment
Type system, ontology

Generalization
Specification/Specialization

Generalization is powerful because it allows you to know things about systems you haven't seen before. It gives intuitive understanding.

We will discuss upper ontologies next. This will inform later discussions on domain ontologies for physics and computer science.

Diagram -> Generalization/Specification -> Top and Bottom/Type Theory -> Generalization is theory building, specification is theory using -> information is both fundamental and very general -> because it is so general, we need to pull from a lot of fields to properly characterize it.

Like branches that furcate from an old oak tree and fork into smaller branches toward the leaves, types spread in a \textit{recursive} manner, each one potentially a parent of arbitrarily many children and an ancestor of

arbitrarily long lineages. As a \textit{type system} like this grows deeper and denser, more information is required to represent it. Similarly, any class in the system \textit{gains information} as it is specified and \textit{loses information} as it is generalized. Thus, a class of type T is \textit{also} of any supertype of T because each possible supertype represents a subset of T's information. For example, the class \textit{Car} is of type \textit{Car}, but it is also of the more general types \textit{Vehicle} and \textit{Machine}.

Naturally, one wonders about what might happen at the extremes of generalization and specification. Three relevant question-answer pairs are given below:

Is there a type so general that it describes everything?
Indeed, there is: it is the \textit{top type} "top," and it describes a special class known as a \textit{domain of discourse} or \textit{universe}. The names speak for themselves---the supertype of \textit{everything} describes the class of \textit{all things} under discussion, the \textit{universe} of relevant topics, the \textit{domain} of our discourse. "top" serves as a useful label in situations when type is unimportant or indeterminate.

Is there a type so specific that it describes \dots\ nothing?
Yes. It is the \textit{bottom type} "bot," and it describes the \textit{empty class}, the class with zero members. It can be considered the type of an \textit{overspecified} class. That is, the class has been defined such that no object in the \textit{domain of discourse} complies. "bot" describes things that are \textit{out of scope} or \textit{nonexistent}. Thus, it is often used as the return type of functions that \textit{diverge} (i.e. do not terminate properly) or as the type of \textit{nothing} (e.g. the 5^{th} element in a 4-element list is nothing).

Are instances just classes that are specified until they have one element?
Technically, no. Specification is a class-to-class process: a class is specified, and a subclass is the result. \textit{Instantiation}, the construction of a concrete \textit{instance} from an abstract description, is instead a class-to-object process. For example, in the above diagram, the class Y_3 is \textit{

```
            instantiated} in order to produce the object b
            (or, by reversing the direction of the arrow,
            we may say that b is an \textbf{instance of}
            Y_3).

Because a type describes a class, it is also incorrect to
   say that $b$ defines its own subtype of \textit{Car}.
   Rather, b is a \textit{token} of type \textit{Car}, a
   physical object that \textit{realizes} one of the many
   possible designs that fall under the class \textit{Car
   }. However, Y\_3 could be specified such that b is the
   sole element of the resultant class {b}. A class with
   one member is called a \textit{singleton class}, and it
    has a \textit{unit type}, a type with one valid value.
    The type of {b} would be a subtype of \textit{Car}
   that describes only b, but neither {b} nor its type are
    the same thing as the \textit{instance} b.

# Notes for Chapter 2: Metaphysics, Natural Philosophy,
   and Physics

A paradigm is a context.
Intro: talk about the long-term trends of science (moving
   away from absolutism, breakthroughs can be made by
   reexamining the foundations)
Endurants and perdurants (substance and form)
Nous is awareness/consciousness/perception

CS is a closed-world.
Time in computer science is often based on the changing of
    state.

## Classical Metaphysics

Wiki: Natural Philosophy

What good is (natural) philosophy in an era of such
   sophisticated science?

What is... metaphysics (study of structure), epistemology
   (study of knowledge), logic (study of truth), axiology
   (ethics and aesthetics, study of value) and
   phenomenology (study of experience)? Make a fun diagram
    maybe?
What is ontology? What is an ontology?
What do metaphysics and ontology have to do with computers
    or programming? Or with OOP specifically?

### Mythology and Cosmogomy

The Universe was described imaginatively before it was
   described abstractly.
```

Religion, mythology, cosmogomy

We're not going to deeply discuss religion or any theory that says more about the supernatural than the natural. However, we will briefly touch on some ancient models of physical reality that are intertwined with spirituality.

Models are constructed within the "religion" of a society. This is not religion in the customary sense, but a spiritual, cultural bias, a set of guiding ideals. Heisenburg calls it the "spiritual pattern of a community." Religion in this sense does not have to be theistic.

"Two kinds of truth"

Religion helps us make sense of the chaos. A common theme is a belief in The One, an interconnection of seemingly disparate things.

The progress of science was held back by war, disease, and also religion. But religion is not inherently destructive. Science and religion discuss two entirely separate things. The mistake was in trying to fit science into a religious model. Too many scientists spent time looking for rigid patterns in the Universe that meshed with their religion, and they just were not there. The Universe is not a divine puzzle that holds the meaning of the divine.

In the same way that there are scientific things that religion has no bearing on, there are religious things that science has no bearing on (ethics). Science will not make religion obsolete. Humans need meaning, community, fulfillment, guidance, etc. Technology does not solve these things. Religion does (however, again, it doesn't require God).

We see a trend from anthropocentric to not throughout human history. We are biased toward our species, our intelligence, our planet, our culture, our language.

The cardinal sin of religion was making something personal into something societal.

Atomic theory of Hindu philosophy

Tao, Yin and Yang

Indra's net

Monad, Pythagoreanism

Classical elements, aether

Pre-Socratic atomist philosophy

And then came Socrates, Plato, and Aristotle

Kosmos and Nous, Plotinus

Gods are a good metaphor for natural events. People wanted to describe something that was more powerful than humans and seemingly eternal.

Polytheistic religions explain the complexity of the world with multiple gods with different personalities and

goals. Monotheistic religions lean toward a god that is benevolent (an omniscient, malevolent god is hard to justify, given that we are still alive).

Religion can serve various purposes: a comfort and a guide in a harsh, confusing world; an explanation of reality and nature; a way to feel one with nature and to find inner peace; a set of rituals that serve to maintain relations between human beings and the tempermental forces of nature; a cultural mythology meant to encourage cooperation between strangers in agrarian civilizations; a promotion of self-discipline and perseverence; an explanation of the vast gap in intelligence between humans and animals; an explanation of subjective experience; an interpretaion of hallucinations induced by schizophrenia or psychedelic drugs; an indulgence in magical thinking and perhaps in the practice of magic; a framework in which one can contemplate personal struggles and express them to a caring parent-figure; a figure to whom one can plead for mercy; a way to process one's own mortality; an assurance that warriors will be rewarded for dying on the battlefield; an interpretation of death such that a future reunion with deceased loved ones is possible; a way to avoid existential crisis and the feeling of meaninglessness; a hope for salvation from this world of suffering; a piece of culture that unites a group of people (and perhaps gives them a sense of superiority); a component of national, cultural, or ethnic identity; an ideological tool used to justify political and social actions; a reason for the congregation of a community; a promotion of nuclear family structure in order to support children; a claim of divine right to rule; a justification for a standard code of ethics; a philosophy that may provide one with a sense of meaning and control.

Sensus divinitatis, sensus deitatis

What are the components of a religion? Mythology, ethics, rituals...

Proto-Indo-European mythology

Shamanism, totemism, animism, spirit world, animal worship and sacrifice

From polytheism to henotheism to monotheism (multiple gods means you need more justification). Dyeus. Aten. Really started in Late Antiquity though.

World egg (Orphic egg), primordial substance, hyle, matter

Son of God, rulers and kings.

Axis mundii. It is natural to associate the sky with heaven because it is bright (which humans think is good), endless, beautiful, and mysterious. Thus, a local people will often consider the holiest place of an area to be the highest one (usually a mountain). This is

the center of the world, and the connection point between Heaven and Earth.
Similarly, familiar places that are close to the center are considered better than the dark, foreign lands that are unknown and scary.

One could argue that philosophy has been done ever since we could think. But philosophy in Ancient Greece was different. Why? Geography. Democracy.
Yes, other cultures had "philosophies" at this time, but the word in this case is referring to a \textit{worldview}. We are discussing philosophy in the sense of \textit{analysis}.

Pre-Socratic Thought: Henosis and Arche

Monism and dualism (and other isms) are just simple, broad models or ways of looking at the world

Orphism. Dionysus (Zagreus), son of Zeus and Persephone, is dismembered, boiled, and eaten by the Titans (who are "white clay men" covered in gypsum). Athena saves the heart, tells Zeus, and Zeus smites the Titans. The resulting soot is mankind, with both the divine soul of Dionysus and the natural body of the Titans. The heart is then placed within a gypsum statue resembling Dionysus. This religion believed in transmigration of the soul, which Plato later supported.
In this story, Dionysus may resemble grapes that are ripped from the vine and boiled into wine (divine + vegetation = wine grapes). In Dionysian mystery religion, raving women known as maenads would get drunk on wine and incite themselves into ecstacy (enthusiasm) by shouting and dancing wildly. The dismemberment and devouring of animals (perhaps including humans) was also associated with these cults. Dionysus was depicted by satyrs, who are half man, half beast. These rituals were explorations of the animalistic side of humans. There was much emphasis on becoming one with the god and phenomenal experience.
Motivation for metaphysics. The One. The Being. The Real. Henosis (the process of unification, unity, or oneness). Henosis is the primordial unification of all things \textit{in perpetuum}, the present and eternal \textit{being} of everything. Henosis is heavily related to the principle of causality. From uncertainty flows the uncaused cause. That is, there is nothing certain at first. Then, something happens, and it causes something else to happen. These events happen relative to each other in space and time.
The One introduces a fundamental question in Western philosophy: Is there an Absolute? Is reality unified in

some way?
The Romans referred to it as the \textit{universus} (literally, the \textit{turned in one}).
Arche means origin (first principle, element, the nature). Philosophy was born in asking what the nature of things was.
Physis (nature, growth) vs. nomos (human convention). The distinction was made maybe because the Greeks had contact with many different cultures interpreting the world in different ways.
Thales. Explaining things without the supernatural. From many things to one substance (the arche is water). Cultural reasons for choosing water. A distinction between appearance and reality. You can live life according to appearances, but you can also learn for the sake of learning. And it turns out learning can be useful, too (olive story).

Heraclitus. The arche is fire because it symbolizes change. Flow of time.
Logos (word, account, principle of order): the great computer program in the sky (or the Word of God, to a Christian).

Empedocles invents the classical elements.
Rebooting Hesiod's Theogony. Instead of gods emerging from the Chaos, let's try to categorize everything naturalistically.

Democritus' Atomism

Time is an instance of the Form of eternity. The Greeks did not think about time like us.
Atoms come in different shapes that give different properties. The shapes also influence sensory perception (bitter/salty taste comes from sharp atoms).
Materialism, mechanism, naturalism

Plato's Theory of Forms

Plato hates Democritus. Wants to burn all of his books. Plato's philosophy is an objection to atomism. Believes that there must be something divine about nature due to its beauty.
Plato is rationalist, Democritus is mechanist and materialist (they agree on epistemology though)
Plato is a substance dualist. The human soul is an element of the Platonic realm.
Essence precedes existence. There is inherent meaning in reality.

Elaborate on why spheres were chosen. The Universe was

also a sphere.

Plato's criticism of Heraclitus' conception of time (wiki Heraclitus)

Plato associates the classical elements with the Platonic solids. He argues that they are not atoms but corpuscles (sort of). Also, the elements are not the arche because they are material.
Plato not very interested in matter. His arche is immaterial (triangles), not material (atoms).
Plato indentified The One with \textit{The Good}. Through philosophy, one uses the Nous to find the truth of the Good. The Good is the Form of Forms. It is the ultimate object of knowledge that allows one to realize all other Forms. It allows us to comprehend justice. The Good is why there is "divine order" to reality. It is the reason why things are intelligible and are not just chaos. The Good is also the reason philosophy and science are possible.

The Good is the Form of goodness. A thing's goodness is its purpose (or telos). Teleology.
The Good is the point of philosophy. There is reason to believe that there is way to exist and think that is good. We can find it with philosophy.
Aristotle's eudaimonia ("good spirit," human flourishing or prosperity).
Pirsig's Metaphysics of Quality
Stoicism

Neoplatonism
The Monad (The One, The Good) is the possibility of existence (dunamis).
The Nous (The Intellect, Mind, Reason) is God or the Demiurge (thinker and composed of thought, divine order, structure instead of chaos). It is an energeia (a thing at work) that actualizes these thoughts as the Forms.
The Soul (Psyche, Pneuma, Breath, Fire, World Soul) is an energeia that actualizes thoughts into existence: it creates a material, living cosmos in the image of the mental Cosmos that is a unified thought within the Intellect (a system of Forms).
Logos is the principle of meditation, which relates the One, the Intellect, and the Soul (the hypostases) to each other.
Eros is ideal love, a love for inner beauty. It is the path to divinity, a desire to possess the Good forever, a desire for immortality.

The Christian Trinity is similar:

The Godhead is the Good (the divine substance).
The Father is the Nous, the Son is the Logos (Word), the Holy Spirit is the Pneuma (Breath).
The metaphor is: there is a Divine Creator (or Mind), and he created everything with the Word (a speech composed by a rational mind) which he executes by means of Breath or Soul (the hot air of life). Just like a Programmer creates a Program that he executes by means of Computation.
Speech scroll

Aristotle's Hylomorphism

Aristotle is sort of a mixture of Democritus and Plato. Not entirely materialist, but not fully abstract. It's scientific, but also teleological. Democritus embraces physis, Plato rejects it, and Aristotle redefines it. Physis is defined by the four causes. Nature and art are both part of physis, but the latter requires an external force/motion. Thus, the natural world has a "soul."
Aristotle did not have a singular arche. But he wrote about minima naturalia, the most fundamental of which were the matter + form combos of the classical elements. He did not believe in a void, reality was continuous (big effect on physics and mathematics).

Substance != matter for Aristotle.
Rejects atomism, says reality is continuous (corpuscles)
Rejects an immaterial fundamental unit, believes substance is real
Four formal causes
Potentiality and actuality (properties of the formal realm and the physical realm, respectively).
Objects are characterized spatially, events temporally.
Mathematical objects and operators
Plato is rationalist, Aristotle is empiricist

"We (collectively, as humans) bestow globality on first principles inductively, promoting the said principles to the status of universal, often missing meaningful alternatives in the process." -- Alon Amit
Put this after Euclid's section?

Theories and Models

Instrumentalism

Patterns

"Models of computation" (coming up in section 3)

Can we say anything objectively about the structure of the Universe? For example, can we say for certain that the Universe is continuous or discrete? Probably not. But we can build some very sophisticated theories and models that describe the Universe very well.

Wiki: analogical models
Medium - mechanical, electrical
Cardinality - N_0 is digital, N_1 is analog
Scope - specific-purpose vs. general-purpose

Axiomatic system -> theory -> model -> simulation

System - a bunch of interrelated parts
Theory - a way of looking at a system (syntactic, system-independent, but perhaps designed with the system in mind)
Model - a way to represent a system (or the application of a theory to a system)

Theoria vs. praxis (considering things outside of our control vs. considering things we can change)
Pythagoras considered theories a way of contemplating math

Scientific disciplines use math to describe parts of reality "in certain terms." Two scientific disciplines might describe the same thing from different vantage points. Theories as models. Scientific laws are not axioms. Models represent reality in some, but not all, ways. If they represented everything, they wouldn't be models. They would be the thing itself. In mathematical models of physics, statements represent physical phenomena.
Theories that describe the same phenomena may differ based on what their basic abstract objects are (particles, waves, bits). These objects are reference points, helpful abstractions inspired by previous observation.
Logic/math theories have axioms/postulates and theorems about formal phenomena and are interpretation-independent. Logic/math models are particular algebraic structures, abstract objects that satisfy a theory (e.g. a set S is a model of set theory, a group G is a model of group theory)
Scientific theories have postulates and laws/principles about natural phenomena. Laws are classical and are expressed with formulas. They are not certain, but are rather extremely probable and backed by mountains of evidence. Principles are modern and are expressed in words. They are "guiding ideas" and do not require proof. Principles can be violated, but you better have good reason to suspect that's the case.
Axioms vs postulates. Essentially synonymous, but the

former is more common in math and the latter is more common in science. Mathematicians do not consider axioms verifiable. In science, postulates are about the physical world, and physics is verifiable, so postulates are verifiable. Postulates can be broken if contrary evidence is found.
Models do not *exist*. They are mathematical abstractions that describe reality. Some phenomenon might be "well-described" by a theoretical model, but that does not make it a signal, field, etc.
Table for model classifications: linear/non, static/dynamic, explicit/implicit, discrete/continuous, deterministic/probabilistic, deductive/inductive/floating
Models may die out if they have major flaws. However, crude models are still sometimes useful due to their simplicity (classical mechanics).
Theories are abstract models, models whose components are symbolic.

A model represents a system, a simulation represents its operation over time.
A computer program can be interpreted as a simulation of some aspect of the Universe and, at the time, as a little Universe in and of itself. The difference between this universe and \textit{the} Universe is that we theoretically have access to all of the information in the former. Whereas we are babies stumbling around in the dark of physical reality, we are gods of the universes we architect in digital memory.
Mindset (blue box?). You can think of programming as just sitting at a desk and typing a bunch of obscure mumbo jumbo. Or you can think of it as the means to simulate anything that you want.

Systems (language systems, logical/formal systems, algebraic systems, automatic systems, hardware systems, software systems, operating systems). All of these can be modeled in terms of information.

Euclid's Elements

Euclid used axiomatic and constructive methods
Axioms come from Aristotle, formal definitions come from Plato

Aristotle's logic was two-valued. However, he did not fully accept the law of excluded middle.

The Greeks used geometry to get around the precision errors of irrational numbers. That's why they used line segments to teach arithmetic.

Elements is great and super important, but in some ways it held mathematics back. It was taught for over 2000 years, and many people assumed it could not be wrong. In fact, it is not considered formal by modern standards, and it is not a general theory of geometry. Critique of the 5th postulate. Non-Euclidean geometry, postulates of special relativity, Galois theory, modern mathematics does not treat axioms as "self-evident" but rather as starting points. Elements assumed many things to be generally true, which are not: numbers are one-dimensional, mathematical objects must be constructed, mathematics must describe the world as we observe it (2d/3d space, concrete objects), mathematics requires straightedge and compass construction, there is only one way to describe geometry. None of these are true, and modern mathematics began when people started questioning Elements.

Aristotle's metaphysics influences science for 2000 years.

Aristotle's ten categories
Moving away from Aristotelian realism, Kant supposes that categories say something about how we conceptualize reality, not about reality itself.

Ancient Knowledge, Lost and Found

And then came post-Antiquity...
Why is it important to talk about this period that is often termed the "Dark Ages"? Well, one reason is that all of the glorious classical knowledge was "lost" for a while outside of Constantinople. Western Europeans had to \textit{discover} the lost knowledge of their ancestors, and they thought very highly of these texts. As we nowadays might opine in favor of "the glory that was Greece, the grandeur that was Rome" (Poe's words, not mine), these peoples revered this knowledge. The trope is common in science-fiction---a lost, ancient civilization, beautiful culture and technology in ruins, far more advanced than what is presently known---but it does appear in history as well, given the right context. Other civilizations (namely the Islamic ones) discovered the knowledge through trade and also valued it, and they built on it in incredibly innovative ways (algebra, combinatorics).
Rome comes and goes, classical knowledge is lost to the West for hundreds of years, but is preserved in Byzantium/Constantinople (Nova Roma)
Christianity dominates the Western world and academic work is done primarily in order to support Christian theology. Naturalism dies out.

Constantine I - lots of Christian theological stuff
   happens in Constantinople.
Some scientific work is done in the West during the Middle
    Ages, but it is isolated and there is little progress.
    Less than 1 percent of the world is literate.
Justinian I and the Plague of Justinian
Charlemagne the Great - the revival of learning
Islamic Arabs seek out and acquire the knowledge of other
   civilizations, Classical Greek knowledge is acquired
   from Constantinople, Translation Movement, the Islamic
   Golden Age occurs, ends due to the destruction of
   Baghdad by Mongols
Europe slowly recovers Ancient Greek knowledge, and
   academic work is done to harmonize Aristotelian thought
    with Christianity
The Black Death happens and wipes out half of Europe,
   scientific work halts
Constantinople falls to the Ottomans in 1453, Byzantine
   scholars flee to Europe, bringing their knowledge of
   algebra
The invention of the printing press allows for the wide
   distribution of information
A major religious schism occurs in Western Christianity (
   Protestant Reformation), prompting rebellion in science
    as well
Copernicus publishes De Revolutionibus in 1543 on his
   deathbed
Galileo champions heliocentric theory and is persecuted by
    the Church for it. "And yet it moves." Galileo's ship.
The Scientific Method (Bacon, Descartes, Galileo, Newton)
   (wiki history of sm)

Particles and Forces

Copernicus, Galileo, Kepler, Descartes, Newton

Wiki - Meditations on First Philosophy (Descartes discards
    all he knows and develops a new metaphysical theory
   from first principles). Powerful statement on the
   importance of doubt.
Descartes publishes the idea of a "frame of reference" in
   1637 (Fermat also discovers it).
Descartes founds the rationalist, mechanistic philosophy
   that early models adhere to (1644). Newton publishes
   the Principia in 1687.
Descartes develops the idea of a system.
Basic assumptions based on observation: continuity,
   determinism, casuality, forces result from collision,
   absolute space and time
Mechanics - study of the behavior of point/body/bodies
   subjected to forces
    Statics - the net force is 0, the net work is 0, the

body does not accelerate (at rest or constant velocity)
Dynamics - nonzero net force, nonzero work, the body accelerates
Kinematics - study of motion of a body without considering cause of motion (no masses, no forces, just motion)
Kinetics - take cause into account

In Newton's era, physics was described in terms of bodies, things with mass (atoms, particles, objects). Scientists followed a mechanistic philosophy (physical reality is simply the motion and collision of matter, the universe is deterministic). Newton was a mechanist as well, but made an exception for gravity, which exhibited "action at a distance." He did not attempt to explain gravity, but simply argued that it matched the data.

Newton discovered that objects would continue their motion if absent of friction, which is a pretty unintuitive idea, considering that there is always friction on Earth.

Optics - Descartes says light is emitted by sources, Huygens develops a longitudinal wave theory of light transmitted though aether, but people don't believe it because it did not explain birefringence, Newton develops corpuscular theory of light and people accept that.

Lagrangian Mechanics
Hamiltonian Mechanics

Blue box for Hume's Problem of Induction and Kant's Trancendental Aesthetics.

Hume states that induction is not infallable and thus is unjustified. Kant argues that while it is not infallable, it is still necessary and rational. Kant backs up the scientific method but argues that it does not constitute an objective truth. Science is not certain, but that's okay.

Critique of Pure Reason - there are other ways of being rational besides deduction

We can only have knowledge of what appears to us, not of things in themselves.

Synthetic a priori knowledge (the knowledge of metaphysics) is what allows us to compute mathematics in our heads. Math is necessary and universal, and math can connect a concept to a new concept other than itself (ampliative).

Religious implications of transcendental idealism

Kant thinks S+T are just human projections. Synthetic a priori. They require no observation and are projected onto surroundings.

Kant: nothing about the object's location describes the object itself.
Walking in a thick grove of trees and finding an overlook into the mountains. Your sense of spatial scope changes. Your mind readjusts.
Define phenomena, noumena, thing-in-itself

Waves and Fields

In early 1800s, people realize Newton's theory does not explain diffraction, interference, and polarization (physical optics). Young and Fresnel develop a transverse wave theory of light. This is still a mechanistic theory, and, thus, it required an aether (as discussed by Aristotle). People tried to understand how an aether could exist if the Earth moves freely through it. Aether drag and wind theories emerge.
Similar theories involving mysterious substances ("imponderable fluids") were formed to explain heat and electricity. Faraday proposes that reality is made up of "lines of force." Maxwell agrees and develops an electromagnetic field theory, but he assumes an aether is required.
Michelson-Morley experiment discredits the aether. Lorentz and Poincare form a theory with a motionless aether that does not interact with matter. In this case, the aether was more of a reference frame. Length contraction and "local time."

Old Quantum Theory and Relativity

"Old quantum theory"
Einstein introduces special relativity, which does not require the concept of aether. Time dilation instead of local time. This theory could be explained in terms of an aether though. The vaccuum of space is more like an aether than a true vaccuum.

Poincare LET quotes

Macroscopic physics is now modeled with relativistic mechanics. It is usually described with Lagrangian/Hamiltonian mechanics/field theories.

Quantum States and Operators

Wiki: Mathematical Formulation of the Standard Model

There was and is a common ethos among professors of quantum mechanics that you should not try to interpret QM as you are learning. The relevent quote comes from David Mermin: "Shut up and calculate!" Many modern

physicists disagree. We will approach it in the same way that we approach everything else in this text.

Modern physics (macro and micro) is described with the Standard Model (quantum mechanics + QFT). This model is imperfect.
Relativistic, quantum, and quantum field models are applied in different situations depending on the scale of space and time involved. Relativistic mechanics encompasses classical mechanics. It's just more complicated, so classical is still used by humans in situations where relativistic effects are negligible. Scientific computing has no reason to use classical though.
Energy is transfered due to colliding particles or radiation. The former is modeled by particle theories, the latter by field theories.
Mass-energy equivalence. Stationary objects have energy because mass is a manifestation of energy. Particles are not necessarily pieces of mass. They are more generally pieces of energy (quanta).

"The principle of the conservation of mass [...] proved inadequate in the face of the special theory of relativity. It was therefore merged with the energy conservation principle-just as, about 60 years before, the principle of the conservation of mechanical energy had been combined with the principle of the conservation of heat [thermal energy]. We might say that the principle of the conservation of energy, having previously swallowed up that of the conservation of heat, now proceeded to swallow that of the conservation of mass-and holds the field alone. - Einstein
Time dilation, causality
The principle of complementarity. Mass and energy (momentum) describe the same thing in different ways. Physics was originally skewed toward the mass view because it is more intuitive and concrete for humans.
In general, outdated models tend to be anthropocentric in scale for space, time, and semantic meaning (theology). However, less general theories are not "broken." They are still useful in certain scenarios (domains of applicability).
At small scales, things are unpredictable. They don't really follow laws. Things happen with certain probability. Classical laws are just averages of tons these probabilities from the same distribution, and thus, at human scale, objects behave basically the same way every time. The probability average is extremely close for each test.
"You can't step into the same river twice." Likewise,

classical physics does not behave in exactly the same way twice. We do our best to show general trends, but the Universe is not certain.

In some sense, quantum mechanics supports object/event metaphysics. Fermions and bosons. Quantum fields of mass and force. Spacetime. Mass-energy. Identical particles.

QM introduces a conflict over scientific realism. Should scientific theories strive to literally describe reality?

Bose and Fermi statistics harkens back to Descartes mind-body duality.

Because QM is probabilistic, this implies that physics is nondeterministic. According to QM, we will never know the true nature of reality with certainty. We cannot know everything about every elementary particle.

Problems with quantum entanglement. With the Copenhagen interpretation, we maintain that models are NOT reality. Just because the math asserts that some real physical quantity has some value, this is not certain without actually performing a measurement. Theories cannot predict values with absolute certainty.

EPR paradox. Bell's theorem. QM seems to say things that are impossible. Throwback to Galileo: "And yet it works."

BIG POINT: It is no longer possible to adhere to both the principle of locality (that distant objects cannot affect local objects), and counterfactual definiteness, a form of ontological realism implicit in classical mechanics. You can only choose one of three: locality, CFD, and conspiracy (that nature only allows researcheer to discover what it wants us to discover). Most people choose locality (because non-locality is contrary to the principle of relativity) and accept that things only exist definitely when they are measured/observed. This is not exactly the same as saying that things are not real, just that we cannot assume that things exist when we are not observing them (e.g. there is no basis in me assuming that Paris exists right now because I am not currently observing it).

Virtuality is perhaps a more accurate word than reality. That is, we can say that the things that we observe have certain virtues, but we can't assume anything more. The phenomenology of synthetic experience is consistent with that of ordinary experience. Virtual reality is "real" in the same way that the world is "real."

"'[W]e have to give up the idea of realism to a far

greater extent than most physicists believe today.' (Anton Zeilinger) .... By realism, he means the idea that objects have specific features and properties - that a ball is red, that a book contains the works of Shakespeare, or that an electron has a particular spin ... for objects governed by the laws of quantum mechanics, like photons and electrons, it may make no sense to think of them as having well-defined characteristics. Instead, what we see may depend on how we look."

Copenhagen interpretation claims that QM is a way of acquiring knowledge, but it does not attempt to describe objects in the way that classical mechanics does. It asserts that founding a model on entities or distinctions (such as particles, waves, and bodies) that are intuitive but ultimately unknowable, leads to incorrect predictions somewhere. Instead, it tries to make claims only about things that can be empirically observed.
Conjugate variables (mass and momentum, uncertainty principle). Derivatives of action. Fundamental interactions.
Beyond the standard model: supersymmetry, integrating gravity: string theory, gravitons

But there is an alternative way of looking at physics that is quite different from the models presented so far...

Information Theory

A pattern as defined in terms of information (Alon Amit). We study patterns in math and use them to model what we observe in physics. All patterns in the physical world can be described by information.

How would you describe a beautiful gown with information?

Bits and bytes and other units of information.
The base-2 metric system (kibi,mibi,gibi,...)
The scale of objects - an essay, a book, an encyclopedia, a 1080p photo, a 4K photo, a 1 minute 1080p video, a 1 minute 4K video, a 1080p movie, a 4K movie, an operating system (Linux vs Windows size), programs (little clients to massive environments like modeling softwares), etc.

If physical phenomena are interpreted as information, this gives formal meaning to the philosophical idea that " we know very little." Much more physical information goes unnoticed than noticed.

Information is similar to energy transfer. It was compared to theories of heat transfer done during the industrial revolution. When you do work, you give some of your energy to something else. The energy may also be interpreted as changing type (kinetic, chemical, etc.). However, this is also just an abstraction. Energy is energy. There are no types really.
For example, when you throw a baseball, you transform chemical energy in your body into kinetic energy in your arm and then you transfer that kinetic energy to the ball. During energy transfer, some of the energy becomes heat. But really, it's all the same energy. You transfer energy to the ball, and most of the energy manifests in motion, but some of it manifests in heat. Efficiency, then, is a measure of how much of the energy involved is doing what you want it to.

Entropy - transformational content of energy
Boltzmann entropy, thermodynamic entropy, von Neumann entropy
Maxwell's demon (the inspiration for daemons in operating systems)

Mathematical theory of communication, Shannon-Weaver model
What does it mean to be a good communicator?

Blue box on what it means to have an information-theoretic view of the world. Give multiple examples that do not involve anything digital.
If you are in possession of information that incriminates you, you want to prevent its communication. You can do this by destroying it (rendering it unintelligible to anyone), encoding it (rendering it unintelligible to people without the decoding algorithm (unless they are code breakers)), encrypting it (putting it in a safe or hiding it), etc.
If you have space or time constraints, you want to use language that is dense with meaning. You can do this with compression in space (removing all of the vowels, using acronyms and Emojis, using shorthand) or compression in time (writing in cursive, no decorations). Often, compression techniques naturally save both time and space (not only do smaller files take up less space, they also download and upload faster).

"The Universe is a simulation." It is possible, but not provable. It is not a scientific question. But thinking of The Universe as a simulation is sometimes useful, and there is no reason it \textit{can't} be a simulation.

Notes for Chapter 3: Information and Communication

Semiotics
Signal processing (analog and digital)
Language
Coding theory

Thinking information-theoretically / computational philosophy

Signals and Systems

Signals are messages composed of time-amplitude data

BB: The Rationale for Binary Numbers
    Implementation details: signals, logic levels, noise, power, reliability
    Close relationship with classical, two-valued logic
    Analog computers have many problems: noise, reproducability
    Ternary electronic computers and their issues
    Binary computers and boolean algebra (logic)
    Beyond binary - fuzzy logic (briefly, discuss more deeply later)

Encoding doesn't have to use numerals. It can use letters, pictures, whatever. You can encode anything anywhere.

Information signals
Information systems

All information is encoded and decoded (digital and sensory)

So, we can represent physical things we experience with digital data. If we collect enough data on the properties of these things, we can simulate reality. We can also simulate abstract mathematical phenomena and other abstract things like language and logic. Much of software design is the simulation of mathematical objects.

Structured manipulation = algorithm
The Shannon-Weaver Model of Communication
Sensory, conversation, and machine examples

We need to communicate an algorithm in a language
What kind of language?
First, let's talk about language in general

Serialization or marshalling - translating an object (abstract or concrete) into a format that can be stored, communicated, and deserialized into an object again (terms are not quite synonymous in Java)

```
Computation with analog data means you are using actual,
   original, non-replicable quantites. Computation with
   digital data means you are using simulated quantities
   that are replicas or model of what you actually care
   about. We model EVERYTHING with numbers: physics,
   language, logic, images, video, music.

Pros and cons of digital and analog computation

One can think of digital encoding as a special case of
   analog encoding, in which a quantity is considered
   analogous to a sequence of digits that represents a \
   textit{number}. Because numbers are abstract and data
   must be concrete, numbers must be associated with
   symbols known as \textit{numerals} if they are to be
   used in computation.

## Semiotics, Language, and Code

Types of sentences - declarative and imperative
An algorithm as an argument

Computation requires communication
Model of communication
All information is encoded and decoded (digital and
   sensory)

Constructed languages, a subset of which are formal
Trade language, lingua franca, Latin
Leibniz - Alphabet of human thought, "best of all possible
    worlds," panglossianism

### Writing Systems

Symbol - syntactic, general term
Glyphs - syntactic, a graphical representation
Characters - semantic, associated with meaning (in
   computing, it refers to an "encoded character" that is
   represented by a glyph), a unit of information (which
   is interpreted by means of a type)
Graphemes - the smallest unit in a writing system, a
   written symbol that represents a sound or phoneme (a
   logogram, a syllabic character, a letter)
Writing systems - ideographic/pictographic systems (Emoji)
   , logographic system, syllabaries, segmental scripts,
   alphasyllabaries

Quipu, number systems
Non-linear alphabets, flag semaphore, animal communication
Character encodings
ASCII
```

```
Westernism - Scientific Revolution, alphabet, character,
   terms and jargon in English (covered in greater detail
   later)
Unicode, code points for abstract characters, UTF-8

### Numeral Systems

and these numbers can be \textit{encoded} with either a
   single \textit{symbol} or a sequence of symbols called
   a \textit{string}. For example, the number eight is
   typically encoded with the symbol 8, which is a \textit
   {numeral} or \textit{digit}. The number could just as
   easily be represented by the symbol \varstigma, but
   this would be non-standard. Numbers are usually encoded
    according to an established \textit{numeral system} of
    some \textit{base} or \textit{radix} that specifies
   how many unique symbols are used for representation.
   Numerical data intended to be read by humans are
   typically encoded in a base-10 system, one of ten
   digits. \\

This design choice is ancient and natural. It is based on
   the ten digits of our hands, which have aided us in
   counting for hundreds of millennia. In fact, fingers
   are the oldest counting tools used by man, and we still
    use them today for this purpose. Other anatomically
   inspired bases have been used in early numeral systems,
    such as base-20, which counts the fingers and the toes
   , or base-12, which uses a thumb on one hand to count
   the bones of the other four fingers. However, today,
   the base-10 \textit{Hindu-Arabic numeral system} is
   universal in mathematics. \\

The base-60 system of Babylonian numerals is also notable.
    Its base is thought to have been chosen for its many
   factors, as this simplifies the process of division.
   Under this interpretation, this system is \textit{
   constructed} rather than natural. Instead of mirroring
   something concrete like physical extremities, this
   system chooses an abstract base in order to make a
   procedural optimization. The reasoning for base-2
   encoding in computing is similar. It sacrifices human
   familiarity for performance and reliability. \\

# Notes for Chapter 4: Logic and Mathematics

This section should cover the logical developments leading
    up to the emergence of metamathemmatics and
   mathematical logic. This includes things like:
Ancient Greek geometry
Aristotle's syllogistic logic
Euclid's Elements
Stoic logic
```

Boolean logic
DeMorgan
Frege
Peirce's Logic of Relatives (relational algebra)

"Of course, the objects of a deductive system are normally thought of as formulas, the arrows are thought of as proofs or deductions, and operations on arrows are thought of as rules of inference."

Laws of thought
Three reliable ways of acquiring information: deduction, induction, and communication.
Hume's problem of induction
Deductive: truisms via consequence, Inductive: facts via evidence

Gottlob Frege
Alfred Tarski
Noam Chomsky
Barbara Partee
Model-theoretic semantics

The Structure of Proof

Proofs and models
String together arguments to make proofs
Tree structure of arguments
List structure of arguments
Data structures structure data in such a way that it can be interpreted as a model of semantic ideas.
Proofs structure text in such a way that...
Information is syntactic, data is semantic

Intuitive Reasoning: The Oracle, the Seer, and the Sage

Seers have premises and conclusions, oracles just speak conclusions
Does intuition involve unconcious, preconcious, or concious reasoning?
Is intuition computation?
This book is as much about intuition as it is about computation.
In some sense, intuition is thought of as the opposite of logic (it has no argument or basis in reality)
In another sense, logical truths (tautologies) could be considered intuitive, so all of logic could be the product of intuition. Unless you believe that logic is objective.
The Oracle and the Seer are unique from the sage because they KNOW their truth is absolute.
Computers are honestly closer to oracles or seers than

sages.
Buddhism - intuition is a faculty of the mind of immediate knowledge

Logical Reasoning: The Mathematician, the Scientist, and the Detective

Deductive and Inductive (+ Abductive)
Mathematicians write proofs (deductive), but they also use creativity (inductive)
Is creativity inductive?
Human computer

Explanatory vs ampliative
Certainty vs uncertainty (probability)
Top-down vs bottom-up
Logical truths (tautologies) vs facts
Abduction = induction + justification
Closed and open world assumptions
Domain of discourse

Can computers induce?
Machine learning simulates induction deductively
Deductive computers, AI, General AI, Artificial Conciousness
Calculation vs Computation
Fuzzy expert systems "abduce"

Notes for the Conclusion to Volume I: Philosophy of Computation

\textbf{Wisdom} is the capacity to think and act in a way that is right. It involves the use of knowledge, a prudent choice of means and ends, and a healthy doubt of one's own understanding. It is born of an acknowledgment of how little one truly knows and how little control one truly has. \\

The wise exhibit:
Realism
Gratitude
Understanding of folly
Humor
Politeness
Self-acceptance
Forgiveness
Resilience
Control over envy
Little regret
Calmness

We tell stories about ourselves through the feedback loop.

Over the course of about 8 years, all of our cells are replaced (true?). Ship of Theseus...
The substance changes, but the \textit{form} remains in our memory.
We feel that we are people who no longer exist, that somehow that the baby in the picture is somehow us. But that baby is gone. Certain forms from its mind live on in our own.

Notes for the Introduction to Volume II: Theory of Computation

Mathematical modeling of computation (sequential, functional, concurrent)
Automata theory covers sequential and concurrent, computability theory covers functional. Together, they cover the prominent models of computations.
Computational complexity theory is a categorization of computational tasks.

Notes for Chapter 5: Automata Theory

Markov chains
Cellular automata
Concurrent models of computation: process calculi, Petri nets, the actor model.
Pi-calculus and ambient calculus

Good idea for a simple automaton: single, married, widowed, divorced legal statuses.

Automata Theory

Wiki: verge escapement
Mechanical computers were born from mechanical automata, which were born from mechanical clocks.

Concept of automata
Models of computation
Computation requires state, transition, and language

From Knots to Nodes

Explain what graphs are so you can use them to model automata.

There are no ends for each knot. A single knot is closed. Links are not like above.

We, however, are interested in the more general \textit{graph theory}, whose focus is on how knots are connected to other knots rather than on the structures of the knots themselves. A \textit{graph} is a drawing

```
   of objects and relations in which each object or \
   textit{node} is represented by a circle and each
   relation or \textit{link} is represented by a line or
   arrow. \\

The string above can be thought of as a list. Here are
   some other structures that are graphs (tree, grid, two
   linked knots)

Node is a joint or a communication hub in an information
   system. It is connected to other nodes by links or
   communication channels. Node is Latin for knot. So node
   /connection in information science, vertex/edge in
   graph theory.

## Concrete and Abstract Machines

Generalize this diagram (Good Automaton) to labelled
   transition system
Transition system is a directed graph

Transition system, semiautomaton, operator monoid
F-coalgebra => transition systems are classes (math object
   )
Difference between semiautomata and automata

The nodes could themselves be automata => automata are
   recursive structures

# Notes for Chapter 6: Computability Theory

Cybernetics

Algorithm characterizations (wiki)

Godel says that Turing's analysis of computability is
   perfect (Interpreting Godel: Critical Essays, p. 118,
   cited in Wiki: Logic in computer science)

Decision problems

## The Scope of Problem Solving

Rewrite -- what are problems, why are they worth solving,
   what is calculation, what is computation?

Domains of discourse / universes in classical logic are
   just types in type theory. Top = true in every universe
   , Bottom = true in no universes (false)

Use sequent notation to explain axioms
Formal systems
Proof calculi (Hilbert)
```

Automata are formal systems

Curry-Howard: proofs and programs are the same thing.

A deductive system is a "calculus"
Argument - set of statements (premises) and a conclusion in a language
Premises can be axioms or theorems
Logical consequence
Truth-bearer
Deductive reasoning is applied to an argument to create a deduction (argument + steps of reasoning)
Composing a deduction is a way to prove validity
Composing an induction is a way to assess strength
Venn diagrams
Reasoning is a set of rules of inference
Inductive/deductive "steps"
Proof - deduction from "known truths" (axioms and proven theorems), proves soundness
Data validation proves cogency?
Argument vs explanation
Validity/soundness

Informal Logic

Discuss the logical reasonings comparatively. 3 paths in life. Certain vs uncertain. Binary vs multi-valued.

Interlocuters
Fallacious reasoning
Formal vs informal (indep of interpretation)

Formal Logic

Common logical symbols wiki
Domain of discourse / universe
Deduction is impossible in non-axiomatic systems because we don’t know what the axioms of the Universe are.

What is logic?

What is early logic?

Syllogistic logic (categorical reasoning, motivated by science 13:45)
Stoic logic

Informal logic is not always explicit. It may have implicit elements that must be made explicit by analysis.
Formal logic, traditionally symbolic logic, now called mathematical logic

```
What is modern logic? ("Arguments and the forms they make
   take")

Hobbes - thought as computation
Boolean logic
Frege, logicism
Class - Russell's paradox

Logical and non-logical symbols
Categorematic and syncategorematic terms (individual
   meaning)
Formal systems
Reason vs. logic -> Humans vs. computers
Model theory

Set theory is intertwined with two-valued logic via the
   membership function

Fuzzy set theory
Membership functions between 0 and 1

## The Ballad of Georg Cantor

Introduce the problem of the "size" of infinity.
Example of the unit circle. The unit circle has a
   circumference of 2pi. It has circumference values of
   [0,2pi] in mathbb{R}. Another way of saying this is
   that it has \textit{central angle} values of [0,2pi] in
    mathbb{R}. The difference between that and, say, a\_n=
   frac{1}{4}\*n, n=[0,7] in mathbb{Z} is that all real
   intervals have an infinite number of possible values.
   But what happens when you go past 2pi, \textit{beyond
   infinity}. You expand the infinite interval\dots to a
   another infinite interval. You'd be hard-pressed to
   find someone who argues that the interval [0,4pi] is \
   textit{not} bigger than [0,2pi]. And yet, the intervals
    have the same "density" (they both have an infinite
   number of values). So the space doubled, and the
   density remained the same. This is not familiar to us.
   "Size" (number of elements) is not a useful measure of
   infinite sets. We need to generalize to cardinality.
Are there infinite sets of different cardinalities? What
   if you doubled your interval an infinite amount of
   times?

Trying to work directly with absolute infinity is trouble.
    Singularities.

### Computable Functions and Computable Numbers

Definitely discuss Principia Mathematica, it's very
```

important step in formalizing mathematics, and Godel proves its goal to be ultimately impossible.
The vicious circle principle

Hilbert's Program and Godel's Refutation of It

Hilbert's Second Problem

Two blue boxes: philosophy of mathematics and Hilbert creating the field of metamathematics with his 1900 speech
Influence of Plato and Euclid on philosophy of mathematics
Mathematical realism, mathematical Platonism
The Big 3 during the crisis: logicism, intuitionism/ constructivism, formalism

Before we discuss G\"odel's landmark theorems in mathematical logic, we must understand a bit about the theory that they describe. G\"odel's theorems on the \textit{completeness}, \textit{soundness}, and \textit{incompleteness} of logical theories are foundational to a discipline in mathematical logic known as \textit{model theory}. \\

Maybe edit above paragraph as you learn more

Most mathematical disciplines focus on a particular mathematical object or structure. An \textit{object} in mathematics is anything that can be formally defined using logic (typically, first-order logic). Given a set-theoretic foundation for mathematics, every mathematical object can be defined as a \textit{set}. A set is a collection of well-defined, distinct objects, and it is itself also an object. Sets can represent objects such as \textit{numbers}, \textit{matrices}, \textit{functions}, \textit{relations}, \textit{points}, \textit{polygons}

Godel's Incompleteness Theorems

If it's true, it's provable.

Modern proof theory treats proofs as inductively defined data structures. There is no longer an assumption that axioms are "true" in any sense; this allows for parallel mathematical theories built on alternate sets of axioms

Verifying a set is recursive by checking that each member can be formed from the axioms (inductively defined)

Recursion vs counting

Godel's thoughts on logical intuition

The Halting Problem

Peano, ZFC

Hilbert's program
ent-shy-dungs-probe-leem
Entscheidungsproblem, which asked whether there was a mechanical procedure for separating mathematical truths from mathematical falsehoods

Church-Turing thesis

Partial recursive functions

Lamda calculus

Turing machines
The halting problem
Something is recursively enumerable when its elements can be counted using recursion. Recursion is a more natural counting method than iteration for sets because sets have no order.

MapReduce is a monoid based alg

BB: Computability is Recursion
    A function can be written iteratively and recursively
    Tail recursion - iterating via recursion
    Some tasks are suited to iteration, others to recursion. What is the deciding factor?
    Recursively enumerable / recursive => computable
    In a well-behaved recursive function, you approach a base case => there is an end in sight for the computation.

Turing Degrees

Posits, chains, antichains
Turing degrees are a join-semilattice
r.e. degrees are dense
Turing degrees of language
Post problem

Notes on Chapter 7: Computational Complexity Theory

Algorithmic information theory, Kolmogorov complexity

Avi Wigderson (2010) proposes mathematical knowability should be based on computational complexity.

```
## Complexity Classes

Time and space complexity are two different kinds of \
   textit{computational complexity}. Computational
   complexity measures how much of a given resource an
   algorithm requires to run. Time and space just happen
   to be the resources we are most interested in. That
   said, computational complexity of \textit{any} kind is
   an algorithmic concept. Data structures themselves do
   not have time or space complexity. The algorithms that
   implement their operations do. However, one could
   informally say that a data structure "has" a certain
   space complexity $S$. That is, if a data structure
   needs to keep track of $n$ values, one could say it
   must store $S$ elements (assuming all of its operations
    are implemented optimally). \\

Some complexities you will encounter in this guide will
   have multiple terms (e.g. $O(m+n)$). Note that if $O(m)
   =O(n)$, $O(m+n)$ could simplify to either $O(m)$ or $O(
   n)$. However, if $m$ belongs to a higher order of
   algorithms than $n$, $O(m+n)$ would simplify to $O(m)$
   (the higher bound). \\

# Notes for Chapter 8: Computer Architecture

## Mechanical Computation

Computation done by humans mechanically with tools
Computation done entirely by machines

BB: Modern Computing is American
    * Discuss the influence of American computer
       manufacturing in the 1950s on the landscape of
       computing today.
    * Things like ASCII, early supercomputer stuff,
       cultural nuances from circles near Harvard,
       Princeton, MIT, Berkeley.
    * British stuff, too, but more on the theoretical side
       .

Hey, let's build machines to do this instead.
Wait, what exactly are we building? Let's model the
   problem.
Computation as a mechanical process vs a function.
Abstract away the mechanics, and a function becomes a
   conversation (input and output).
Computing is more than just math, semantically.
Structure of language, types of sentences (declarative/
   imperative)

Calculation vs computation. Separate histories.
Is a microwave a computer?
```

Modern cars are full of computers. Embedded systems.

We've discussed "computers" in many different ways (primitive examples, pre-Turing examples, theoretically).
The typical modern computer is programmable (in a Turing-complete sense), digital, electronic, stored-program, general-purpose, uses RAM, uses so-and-so architecture
Maybe some ubiquituous devices and "optional" components (like hard drives and I/O)
Maybe discuss some important non-standard computers (embedded systems, datacenters, supercomputers).

Signals, clock cycles
Byte vs word addressing

Notes for the Conclusion to Volume II: Theory of Computation

Notes for Introduction to Volume III: Types, Structures, and Algorithms

This is the mathematics chapter.
Look into the following:
Abstract algebra
Type theory
Category theory
Algorithmic information theory (Kolmogorov complexity)
Chaos theory (nonlinear systems)

"My math background made me realize that each object could have several algebras associated with it, and there could be families of these, and that these would be very very useful." - Alan Kay

HUGE POINT: An object consists of a type and a structure. A type is what the object is semantically. What does the object mean? A structure is what the object is syntactically. What is the object made of and how is it arranged?

Structures, not objects. For example, we are directly interested in list structure and only indirectly interested in objects *with* list structure. We focus on Forms, not on complex particulars.

An abstract structure of fixed information content takes up a fixed amount of computational space. What is the difference between a structure and a space? A structure is a space with an associated type that represents constraints. A structure is a particular kind of space. (Look at quotes on space wiki).

Why leave set theory? We want to explore proper classes, so we generalize set theory to category theory. Sets become objects and functions become morphisms.
Type theory is a formal syntactic language of calculus for category theory. Category theory provides semantics for type theory.
A mathematical object in something that exists in a mathematical universe. They can have many different kinds of structure. We are interested specifically in the \textit{algebraic structure} of objects.
An automaton is a mathematical object
Are objects and relations fundamentally different?

Notes for Chapter 9: Type Theory

Intentionality and extensionality - types are intensional, classes can be either (wiki - Class (philosophy))
In this case, class is equivalent to the idea of a set in mathematics. Sets can also be defined extensionally and intensionally.
In mathematics, the word class has different definitions based on which foundation of mathematics you are working in. Informally, in mathematics, a class is a collection of mathematical objects. In Zermelo-Fraenkel set theory, a class is informally a collection of sets (which are formally defined, axiomatized). In Von Neumann-Bernays-Godel set theory, the word class refers to the abstract idea of a proper class, which is formally defined. In ZFC, sets are elements of other sets, but classes are not elements of any other class (e.g. the class of ordinal numbers is not a set). In ZFC, sets are objects in your universe, classes are not. Sets are semantic objects, classes are syntactic objects.
Before the foundational crisis of mathematics, the word set was used as an informal term for a collection of things. Set theory formalizes the definition of a set. Cantor's and Russell's paradox revealed that not all collections are sets. A class is any object that can be described in the langauge of set theory. Not all classes are sets. The axioms of set theory apply only to the sets.
A category in category theory is a pair of classes (basically). In this case, the word class means either set or proper class. If both of its classes are actually just sets, the category is \textit{small}. Otherwise, it is \textit{large}.

Essence - philosophy
Curry-Howard Isomorphism
Not all data structures implement abstract data types.

Imperative vs functional ADTs

Talk about how relations are sets (objects)
Functions are deterministic relations

In compsci, Method (can be abstract) vs effective method (steps fully described and finite, can be written in pseudocode) vs algorithm (steps fully described and finite, can be written in pseudocode, calculates the values of a function) vs an implementation of that algorithm. Church-Turing theorem, computability theory and what not... Talk about how every iterative function can be written recursively and vice versa. Any time you use recursion, you're basically adding a stack to the algorithm.

Contiguous or non-contiguous? Array or node?
Mention other ADTs that a DS could implement
Address lack of pictures. Add pictures for ADTs?
Delete operation should acutally be remove. Discuss the distinction.

The Curry-Howard-Lambek Isomorphism

Recursive composite type = a dynamic data structure

Curry-Howard Isomorphism

Logic programming

Monoids, groups, algebras, coalgebras, etc.
Signatures

F-Coalgebra
Semiautomata and transition systems
Algebraic specification

Notes for Chapter 10: Algebra and Category Theory

Notes for Chapter 11: Abstract Data Types and Data Structures

ADTs are "types" of metaphysical objects.
These are just patterns that appear in life. Trees tend to be more common in nature, lists are how humans tend to organize things.
Do "plain old data structures" first: arrays, structs, records, unions, objects
Data type + data structure wiki

What is a module? What is a dictionary?

The subsections should be ADTs. The subsubsections should be data structures. If you really want to discuss a

```
   specialized implementation, you can make it paragraph
   or a blue box. But you should seriously consider
   pushing the implementations into an appendix for better
    thematic cohesion.

What do I mean when I say... type (or data type),
   structure (or data structure), object, instance, class
   (of objects), implementation
If an element has type $X$, it belongs to a class $X$. We
   typically speak in terms of types in theoretical
   computer science (harkening back to Alonzo Church) and
   in terms of classes in object-oriented programming (
   harkening back to the programming language Simula,
   whose designers Ole-Johan Dahl and Kristen Nygaard were
    harkening back to the idea of a class in philosophy
   and mathematics).

Iterators

## Lists

Talk about tuples, how they are different than lists, and
   why the preferred term is list (streams, countable
   infinity)

### Arrays

Talk about how strings are often implemented with arrays.

### Linked Lists

An interesting example - The Undo & Redo text editing
   process is a linked list.

## Graphs

Flow networks
Graph coloring
Minimum spanning trees

## Trees

Talk about a "single branch tree" or a "path." Kind of
   like a linked list, right? Basically, there are two
   design philosophies to data structures: contiguous and
   non-contiguous (or node-based).
Discuss the array implementation of a binary tree.
Talk about how file systems are trees
Talk about how this guide is tree-structured
   (3->3.1->3.1.1)

# Notes for Chapter 12: Algorithms
```

Algorithm characterizations (wiki)

Precomputation: Searching an unsorted array is O(n). Sorted, it's O(logn).
BUT, if you do O(n) work upfront, you can place the array's elements in a hash table to get an O(1) search time.

Notes for Conclusion to Volume III: Types, Structures, and Algorithms

Notes for Introduction to Volume IV: Computer Programming and Operation

Notes for Chapter 13: Programming Language Theory

Programming is the act of scheduling something (e.g. cable television programming is the strategic scheduling of different kinds of TV shows). Computer programming is the act of scheduling machine operations, either one by one or in blocks of operations, which are abstracted away and represented by statements written in high-level languages.
Programming is also the act of optimizing (i.e. maximizing or minimizing) a function in order to predict the best course of action. In computer programming, this involves developing input (source code) that will perform a task with minimal complexity (in space or time).

Primitives (derived from primitive notions in philosophy)
Virtual machine
REPL
Entry point
Interactive programming
Programming in the large and programming in the small
Pipeline
Transcompilation
Command line interpreter
Modular programming
Template processor
Software framework
Portability

Clocking, asynchronous events

Iteration and recursion are about repeating an operation. How they do that differs.
Iteration: fast, usually easy to implement
Recursion: easy to debug, sometimes more natural, uses stack memory
They can be converted into each other

Polyglot programming

Elements of Programming Languages

Development: specifications should be separate from implementations

What does a Turing-complete language need? Conditional branching (not just jumping), others?

Structured program theorem - any computable function can be represented using three types of control structures (sequence, selection, repetition)

Statement types: assertion, assignment, goto, return, call

Field, property, method, object, class, ...

History of Programming Languages

What does it mean to print? It means to render a piece of information content in some physical medium. Printers print information into books. Back in the day, when computers didn't have fancy graphical displays, the print command converted digital content (information stored in bits) into a print file or device file, a file that an external device (like a printer) can read and operate on. The operating system of the computer would then output the file to the device, which would render the information content in physical form. Nowadays, the print command converts digital content into various standard file formats and those files output graphically by means of a monitor (and technically, light is physical). Printing hasn't fundamentally changed.
Turing is imperative and lambda is declarative?
What is a paradigm? How does having a strict paradigm affect the experience of coding? More restrictive, more streamlined?
More recent is not necessarily better
Why are we still using old languages like C?

Programming Paradigms

How can we categorize programming languages?
Imperative vs. Declarative (Functional and Logic)
        Declaration-style and expression-style
Procedural vs. Object-Oriented
Compiled vs. Interpreted
Static vs. dynamic
Weakly typed vs. strongly typed
Statically typed vs. dynamically typed
Syntax (sparse, dense, graphical)

Release date
Structured vs. Non-structured (Control flow)
Popular use cases
Cultural perceptions?
Company preferences (Microsoft, Apple, general use)
General-purpose vs domain-specific (MANY domains)
High-level vs. low-level
System programming vs. application programming
Scripting languages (glue, shell, macro, embedded)
Concurent/Parallel/Distributed/Multithreaded
Heterogenous programming (OpenCL)
Natural languages
Memory management vs. garbage collection (C++ scope-based
   resource management)
Performance vs. readability/manageability
Expressive power
Whitespace-sensitive or not
Data languages
Query languages
Pointers or not
Exceptions or not
Library size
Numerical computing (e.g. MATLAB, Maple, Mathematica)
Automated theorem proving
Hyperlink to esoteric languages lol
Literate programming (e.g. TeX)
Ontology language

Programming Techniques

Higher-Order Programming

Map, filter, fold, zip

Notes for Chapter 14: Specification and Implementation

Language vs implementation vs platform
Dependency injection

Notes for Chapter 15: Operating Systems

This chapter should focus on the theoretical side of
   operating systems (memory management, threads, locks,
   hypervisors, etc.). However, this content is
   inseperable from its history; these concepts were
   implemented as improvement to early, real-world
   operating systems. Nevertheless, a distinction is drawn
    between this "operating systems theory" and the
   practical, everyday use of an operating system in an
   efficient manner, perhaps as a means to write software.
    The latter content is found in the next chapter, "
   Practical Computing."

Teleprinter, teletype, TTY
Terminals: hardware terminals and psuedo-terminals (https://unix.stackexchange.com/questions/93531/what-is-stored-in-dev-pts-files-and-can-we-open-them)
Terminal vs. console (root)

Multics -> Unix -> BSD -> GNU -> Linux

Here is what an operating system is. Here is the distribution of modern operating systems. These operating systems are called Unix-like because they are built on top of or heavily influenced by the operating system Unix. Variants of Unix include: BSD, GNU/Linux, macOS, etc. Non Unix-likes include: VAX/VMS, OpenVMS, DOS, OS/2, Microsoft Windows (which is a family of OSs)

In 1969, AT&T created Unix. It was originally written in assembly, but was rewritten in C by Dennis Ritchie in 1973.
AT&T lost an antitrust case, preventing them from getting into the computer business. As a result, they had to license Unix to anyone who asked. Unix became very popular as a result because it was a revolutionary OS. Unix is, however, proprietary.
In 1984, AT&T divested Bell Labs, and Bell Labs started selling Unix.
Richard Stallman started the GNU project in 1983 to create a free, open-source, Unix compatible OS. The project started with the programs rather than the kernel.
Linux is basically open-source Unix. It was written from the ground up by Linus Torvalds in 1991. It is based on Minix (mini-Unix), which was an academic, microkernel OS based on Unix. Linux is a monolithic kernel.

Blue box on privacy
Software law
Free Software Foundation
FOSS/FLOSS and Open Source
Binary blobs (proprietary device drivers)
Shoshana Zuboff (The Age of Surveillance Capitalism)

Computer networks
The Internet
Web 1.0, Web 2.0, and Web 3.0 (the Semantic Web)

Bare machine - a computer executing instructions without the aid of an operating system
Environment variables ($ indicates that the (all-caps) name following it is a variable; env variables are exported from cmd)
What is a shell? Why are there so many of them?

Unix philosophy

Notes for Chapter 16: Practical Computing

What is computing? What does it mean to compute?

Good computer habits are about: controlling what
   information you receive on a daily basis, performing
   everyday tasks in the most efficient way possible,
   being intentional, ...
I'm going to be honest: most people's computer habits are
   terrible. I'm going to go out on a limb, dear reader,
   and say that \textit{your} computer habits are terrible
   .

Application vs. program. Basically the same. Applications
   are programs that run on an operating system and help
   the user complete tasks. Some system programs are not
   applications.

The importance of doing things the hard way. Struggling at
    first so that you may reach higher heights than those
   who choose the more accessible tools. The importance of
    restricting yourself to a few simple tools.

What is data loss and how can it be prevented?

Linux

Real programmers use Linux. Unless they work for Microsoft
    or Apple. But Linux is a demonstrably better OS for
   people who are serious about computing.
TUIs (textual/terminal-based user interfaces) / CLIs (
   command-line interfaces) are better than GUIs (
   graphical user interfaces). Text is pure content (words
). Graphics are appearances (signs). The number of
   possible signs is much greater than the number of words
    in any practical language. For practical computing, we
    want things to be as simple as possible. Graphics
   should only be used when necessary (browsers, pictures,
    videos). Symbols can be used in very simple cases (a
   music player: play, pause, previous track, next track).
    But computing is more complicated than that and
   requires a language. Text is the better fit.
TUIs also prevent repetitive strain injuries and promote
   home-row typing, which improves the mind-muscle control
    of your ring and pinkie fingers.
Corollary to the above: \texttt{vim} is better than \
   texttt{emacs}.
Tile-based WMs are better than drag-and-drop (a.k.a.
   stacking or floating) WMs. Drag-and-drop allows too
   much freedom. How often do you need a window that isn't
    fullscreen, 1/2, or 1/3? Apple is responsible for the

"desktop metaphor" and the drag-and-drop trend.
Using Linux via the command-line makes me feel like I am *operating* a computer. It is the right way to compute.

What is a distribution of Linux? A collection of software based on the Linux kernel that forms an OS.
Typically includes: Linux kernel, package manager, utility software (such as the GNU utilities), window system, window manager, desktop environment.

What is Arch Linux?

File Systems

Unix filesystems: zfs, js, hfx, gps, xfs

Run tree / -L 1 and run through that list alphabetically, explaining symlinks
"Binary file" or "binary": an executable program

home is represented with ~ in UNIX because home and ~ shared a keyboard key on the ADM-3A terminal
/usr originally meant "user" or "users." It may also be interpreted as the backronym "Unix System Resources" or "User System Resources."
/usr/bin: binaries that come with the OS or binaries that are installed by the OS's package manager
/usr/local/bin: binaries that are installed locally (by hand, by the user on the machine itself). This is where you should store the scripts that you write.

Naming conventions for files and directories:
1. Case-sensitive
2. Upper and lowercase letters, numbers, dot, dash, and underscore allowed (special characters are also technically allowed but must be escaped). \\0 (NULL) and / are absolutely forbbiden.
3. File extensions are optional
4. Files that start with a dot are hidden.
5. Names must be unique inside their directory
6. 255-char limit
Conclusion: the names of files/directories should only use lowercase letters, digits, and either underscore or hyphen (user preference). Other options are just not practical.
Exceptions are allowed for some files (Java classes start with capitals).
Spaces are not a good option for file names because spaces separate command arguments. You can use them if you are fond of the aesthetic, but you will have to enquote your files or escape the spaces in their names.
I prefer kebab case. Just don't start the name with a

hyphen because that implies that the name refers to an option.
my-thing is a file. my-thing/ is a directory.

Colors can be handled in one of two ways:
1. set your background to a single color (or perhaps a stylized background that utilizes a very simple palette of colors) and then download a complementary theme for your terminal
2. find a really cool wallpaper with lots of colors and then use pywal to generate a theme for your terminal (which can be hit or miss)

Fonts
TTF vs. OTF

Common Commands and Tasks

How do commands work? Arguments, options, parameters, subcommands...

When to write an alias vs. function vs. script?

Common tasks on a computer:
Reading and writing
Downloading and uploading (i.e. writing a remote file to local storage and writing a local file to remote storage)
Streaming (i.e. reading a remote file)
Cut/copy/paste (or, in Unix, delete/yank/paste)
Logging in and signing off
Start/kill/list processes
Highlight text (or, in Vim, visual select)
Navigating a filesystem

Superuser vs user

Connecting to WiFi:
nmcli dev wifi list
nmcli dev wifi con 'SSID' password 'password'

Download a file:
wget "<address>"
wget -P <location> "address"

Kill a process:
ps aux | grep <process-name>
a = show processes for all users
u = display the process's user/owner
x = also show processes not attached to a terminal
kill <pid>

File permissions: read (r), write (w), execute (x)
10 chars, positions 0-9
Position 0: file (-), link (l), directory (d), etc.
Three triads: user (1-3), group (4-6), and outsider
   permissions (7-9)

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

Userspace

What is ncurses? Important?
Install necessary fonts (Unicode, Emoji)
Use feh to change your wallpaper

What is a...
    desktop environment?
    window manager? (xMonad)
    login manager? (startx)
    package manager? (pacman)
    package repository? (Arch repos, AUR, GitHub)
    terminal emulator? (st, alacritty, kitty)
    text editor? (neovim, emacs)
    status bar? (polybar)
    compositor? (compton)

Everyday Tools

Essential Programs

Wiki: Utility software

Pick a good one for each task and stick with it. Learn
   about the program and get good at it. When you get

```
   really comfortable with it, you can consider exploring
   other programs.

Shell (bash)
Terminal emulator (st)
File manager (ranger)
Desktop environment (none)
Window manager (i3)
Process manager (htop (htop-vim-git), gotop)
Power management (acpid)
Screen locking (to get: i3lock or slock)
Application launcher (dmenu)
Text editor (vim) (IDEs are not necessary)
Typesetting system (texlive: includes tex and pdftex
   programs, latex and context macro packages, and the
   xetex (a.k.a xelatex) and luatex engines; edit with
   vimtex)
Browser (firefox)
Torrent client (qbittorent)
Notes (convert markdown files into pdf: pandoc <*.md> -s -
   o <*.pdf>)
Image viewer (imv (feh for setting wallpaper))
Document viewer (zathura)
Image manipulation app (gimp)
Music player app (cmus)
Video player app (vlc)
Games launcher
Linter (to get: ale)
Debugger
Tester

What is streaming? What is a streaming client?

Home is where the user lives. His belongings are usually
   categorized according to media type. Documents are for
   text, Downloads are for packages and zips and tarballs,
    Music is for music, Pictures is for pictures, Videos
   is for videos.

What is text? (EOF character)

Important shell commands and coreutils:
ls (plus la and lsd)
ln (create a link)
cd
find

pandoc - a universal document converter

Vim:
Geneology: ed -> em -> ex -> vi -> vim (-> neovim)
If you never learned how to touch-type, vim will teach you
```

.
:h E<number> (error help)
Plug-in manager = vim-plug
vimtex for latex
Tabs, not spaces! Modify .vimrc to make the tab key insert
    4 spaces instead.
Document tabs in vim: :tabe, :tabc

Version Control

Git

How do you write a good git commit message?
It should give context to your changes.
It should be in the imperative mood.

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

* Clone and install via Makefile: git clone the directory
   (/var/git is a good choice), make && sudo make install
   (compiles and stores an executable in /usr/local/bin).
* Download raw files with wget, only a good idea if you
   just want a few files
* Get it from an official package repo (the Arch repos)
   with your distro's built-in package manager (pacman)
* Get it from an unofficial package repo (the AUR) with a
   package manager that you choose (yay)
* Manual installation: Download a tarball, unzip it,
   extract it, and build the source files into an
   executable

Languages and Language-likes

The Internet: HTML5, CSS3, PHP, JavaScript

Notes for Conclusion to Volume IV: Computer Programming
   and Operation

Notes for the Introduction to Volume V: Software
   Craftsmanship

Wiki: continual improvement process, kaizen, genchi
   genbutsu, The Toyota Way

http://www.openprojects.org/software-definition.htm

Compsci vs software engineering
What is the right term? Software developer, engineer,
   architect, designer? Programmer?
Architecture metaphor
The Church and the Bazaar

CRUD

Systems development lifecycle
What is frontend, backend, and full stack?

Notes for Chapter 17: Software Engineering Processes

Program life cycle

Agile
Test-driven Development (TDD)
Single Responsibility Principle
Open-closed principle

Blue box on categories of software engineering questions: pure DS&A problems (anything that sounds mathy), word problems (anything that sounds techy), tool familiarity (knowledge of software principles), tool proficiency (trivia, basically)

Notes for Chapter 17: Design Patterns

Singleton
Abstract Factory

Notes for the Conclusion to Volume V: Software Craftsmanship

Final words (what I want you to take away from all of this):
The importance of context and judging things relationally.
The importance of history and of earnestly considering the thoughts of those who came before you.
Respect for the power of information. Speak carefully.
There is nothing so complicated that you cannot understand it. If there is, it is something that no one will ever understand. Thus, it is discipline and perseverence that matter, not intelligence.
You should strive to think logically. Otherwise, you will spread misinformation and skew the models of others.
Now that you possess the basic tools of creation, you must use them wisely. Do not create without first considering the consequences of doing so. Only build what is good.
If you are uncertain, take some time to think. The answers lie in logical consistency and a proper evaluation of all of the factors.
There is an immense amount of information out there today. There is, however, very little knowledge and a dearth of wisdom.
The overview effect