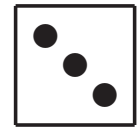


Understanding Computation

What is computation? How is computation different from calculation? What kinds of tasks can computers perform? To provide a foundation for what makes quantum computing special and to ground our policy analysis, this chapter visits the history of computing, starting with the ancients and their concepts of mathematical concepts, and proceeds to discuss modern classical computing.

Humans have been using numbers since at least ancient Sumer and Babylonia, five thousand years ago. The Babylonians were fascinated with the number 60; they thought that the number 60 was mystical, since it could be divided into two, three four, five, six, 10, 15, 20 or 30 pieces. But for the majority of human history, manipulating numbers was something done by people, not machines, sometimes with tools such as the abacus, but more capacity was needed.

When machines took over the task of manipulating numbers, it was often because of war or military efforts. Designing and building these machines took government funding, often supplemented with support from private companies and brainpower from academia. These facts are stressed here because just as early analog computers were electromechanical engineering marvels, building quantum computers requires state-of-the-science engineering at particle-level scales, with experts from several disciplines, and the funding to match. Also emphasized is how computers can be miniaturized, be reproduced for a fraction of their initial costs, and find their way into everything, including even doorknobs. Computing can enjoy a virtu-

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

ous cycle where simple devices can reveal efficient design for even larger, faster computers. This insight will be key for the trajectory of quantum computers.

This chapter also introduces complexity theory to explain the kinds of problems that are hard for computers to solve. This lays the groundwork for understanding the different capabilities and potential advantages of quantum computers. This background is crucial to understanding quantum computers for two reasons. First, it dispels the common notion that quantum computers would be a kind of magical device that can ponder all possible solutions to a problem. Instead, quantum computers, like any other kind of tool, are good for some tasks but no better than ordinary computers for others. Second, complexity theory helps illuminate what is truly exciting about quantum computers (hint: it is not whether encryption can be cracked). Instead, if quantum computers can solve problems out of reach for classical ones, quantum computers will help solve some of the difficult, costly challenges in life. Complexity theory helps elucidate the kinds of efficiencies that could come about, from finding ways to optimize energy-intensive processes to finding valuable information in enormous datasets.

This chapter should be read by those who need to make investment decisions or otherwise understand the underlying technology and assumptions. This chapter lays the groundwork for understanding what quantum technologies are likely to do and, conversely, helps identify the specious claims so often made about the capabilities of quantum computers.

3.1 Mechanical Calculation

Machines are systems that use multiple parts and some kind of power for performing some kind of task. "Shovels are tools; bulldozers are machines," we are informed by Merriam-Webster.[1] Machines are different from tools in their complexity and their power. The earliest known calculating machine is the Antikythera Mechanism, a device with more than 30 interlocking bronze gears that was found in a shipwreck off the small island of Antikythera, Greece. Although the user's manual for the mechanism did not survive, this 2000-year-old mechanism has now been thoroughly reverse-engineered and is be-

[1] Merriam-Webster Incorporated, "'Machine.'" (2020).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

lieved to be a means to predict the movements of the planets and the occurrences of eclipses.[2] You can even download a simulator.[3]

The Antikythera Mechanism used differently sized wheels with teeth to account for the differing speeds of the planets; a peg that cycles back and forth in a slot accurately represents elliptical motion of the Moon, which is attributed to the Greek astronomer Hipparchus of Nicaea (*c.* 190–*c.* 120 BCE). The mechanism thus implements a kind of multiplication, but the ratios were set and unchangeable, like the motions of the planets themselves.

It took another 1700 years before the basic building blocks of flexible mechanical calculation were put into place. In the early 1600s, the Scottish mathematician John Napier invented two approaches for multiplying and dividing numbers using addition and subtraction. The first, called "Napier's bones," embedded numeric tables on wooden rods. The second and more powerful approach used logarithms, which Napier also invented. Napier published the first book of logarithms in 1614. Sixty years later, the German mathematician Gottfried Wilhelm Leibniz (1646–1716) started working on a mechanical calculator that could add, subtract, multiply and divide when the user set dials to various positions and turned a crank. Critical to this invention was what is now called the Leibniz wheel, which causes the dial that shows the tens' place to advance from "0" to "1" when the dial showing ones advances from "9" to "0." In 1820, the French inventor Charles Xavier Thomas de Colmar (1785–1870) introduced the Arithmometer, the first commercially produced mechanical calculator: his factory built a thousand of them before his death in 1870. Meanwhile in England, Charles Babbage (1791–1871) designed the world's first automatic calculator in 1822 for the purpose of calculating and printing tables of logarithms, trigonometric functions, and artillery tables. Babbage called his invention the "difference engine," and obtained funding from the British government to build it in 1832.

Although all of these devices proved to be helpful aids to humans performing tasks involving numbers, none of them could *compute* in the modern sense. That's because they all lacked the ability to alter their computations based on the results of a specific calculation. This is what distinguishes a machine that *calculates* from one that *com-*

[2]Spinellis, "The Antikythera Mechanism: A Computer Science Perspective" (2008).

[3]Goucher, "Antikythera Mechanism" (2012).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

putes. Babbage realized that the difference engine was limited, and designed an improved system he called the analytical engine. Alas, Babbage never built his invention, although a group of enthusiasts in England called Plan28 are now working to do so. You can follow their efforts at plan28.org.

3.2 The Birth of Machine Computation

Babbage may have seen the future, but there is no clear evolutionary descent from his machines to the computers of today. Instead, the first computers of the 1940s descended from the invention of punch cards and card-sorting machines that were developed for the 1890 US Census. The invention of teleprinters and punched paper tape was a way of making more efficient use of telegraph lines, and to manage the growing demands of science, engineering, and various militaries to perform increasingly complex numerical calculations.[4]

World War II saw two significant efforts aimed at using automated calculation for the war effort. There were two radically different applications for automated calculators, with the United Kingdom leading the development of machines to solve combinatorial problems, and the Americans largely developing machines to solve numerical ones.

3.2.1 Combinatorial Problems

In the United Kingdom, a project headquartered at Bletchley Park developed a series of hard-wired special purpose devices for cracking the German military codes. Cracking those codes is a "combinatorial" problem because the encrypted text was created with a "key" represented by the complex (for its time) initial settings of German encryption devices. The goal of the project was to determine which combination of those settings produced the encrypted text sent by the Germans. This is the project on which Alan Turing worked, and which is featured in the somewhat factual Hollywood film *The Imitation Game*. Initially this project used electromechanical devices called "The Bombe" to search the possible settings for the Germans' Enigma encryption device. In the movie there is a single Bombe, but in reality there were hundreds of them, each one working on a different part of the problem, or a different encrypted message.

[4]While there are many histories of computing, we recommend the eminently entertaining coffee table book by Garfinkel and Grunspan, *The Computer Book* (2018), as well as the more scholarly book by Dasgupta, *It Began with Babbage: The Genesis of Computer Science* (2014).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

As Copeland explained it:

> "The Bombe was a 'computing machine' – a term for any machine able to do work that could be done by a human computer – but one with a very narrow and specialized purpose, namely searching through the wheel-positions of the Enigma machine, at super-human speed, in order to find the positions at which a German message had been encrypted. The Bombe produced likely candidates, which were tested by hand on an Enigma machine (or a replica of one) – if German emerged (even a few words followed by nonsense), the candidate settings were the right ones."[5]

The second code-breaking project at Bletchley Park – one that was shrouded in considerably more security – used vacuum tubes to crack the military codes used by the German High Command. Tubes can switch electrical circuits 500 times faster than relays. This complexity was essential, as the encryption machine developed by C. Lorenz AG had 12 encryption wheels, compared with the three or four used by the Enigma. The system was called Colossus, and the UK only built ten of them. The engineering on these systems was fantastic. For example, input data was on punched paper tape, and the computers were so fast that the paper tape had to move at 35 miles per hour. The Colossus computers did their job so effectively that all were destroyed or dismantled at the end of the war in order to protect the secret of the UK's code-breaking capabilities – a secret that it kept until 1974, when F. W. Winterbotham published his book *The Ultra Secret*.[6] A similar code-breaking effort in the US called Magic was under the direction of William F. Friedman, at the US Army's Signal Intelligence Service, the precursor to the US National Security Agency. The US story of how early punch card tabulators from International Business Machines were modified to perform cryptanalysis has also been told,[7] but it is not as well known as the story of Bletchley Park.

[5] B. J. Copeland, *Alan Turing's Automatic Computing Engine* (2005).
[6] Winterbotham, *The Ultra Secret* (1974).
[7] Rowlett, *The Story of Magic: Memoirs of an American Cryptologic Pioneer* (1999).

3.2.2 Numerical Analysis

Digital computers were also under development by the US military, but on the western side of the Atlantic the generals wanted to solve numerical problems, rather than combinatorial ones. Specifically, the military was seeking solutions to differential equations.

The military's interest in calculus was a direct result of improvements in firepower.[8] In 1800 the range of a big gun on a naval vessel was only 20 to 50 yards, making artillery pretty much a load, point and shoot affair. By 1900 naval guns could reach 10 000 yards: scoring a hit on an enemy ship, or a target on land, required accounting for the speed of the firing platform; the speed, direction, *and temperature* of the wind; the weight of the shot and the amount of propellant; and even the rifling of the gun's barrel. Spotters looked for splashes with precision optics, measuring (to the best of their ability) the distance and direction of the misses. All of these factors were used to calculate the azimuth, elevation, and amount of propellant used in the next shot. Artillery had become highly mathematical.

In 1927, an MIT professor named Vannevar Bush began work on a mechanical device that could evaluate calculus integrals and other kinds of mathematical function using a combination of spinning rods, gears, wheels, and several metal spheres. Bush, who became MIT's Vice President and Dean of the School of Engineering in 1932,[9] knew that the machine had both scientific and military applications. Specifically, the machine could be used to simulate many slight variations of the trajectory of an artillery shell, making it possible to produce numeric tables that could be used at sea (or in the field) by gunners to target their artillery faster and with more deadly precision. Bush originally called the machine *a contin-*

[8] Clymer, "The Mechanical Analog Computers of Hannibal Ford and William Newell" (1993).

[9] Vannevar Bush went on to become president of the Carnegie Institution of Washington, a philanthropic research funding organization in 1938. He soon became chairman of two US government agencies: the National Advisory Committee for Aeronautics and the National Defense Research Committee, effectively making him the US government's chief scientist. Bush initiated the Manhattan Project and convinced President Harry S. Truman to create the National Science Foundation (NSF), which was signed into law in 1950. Today he is frequently celebrated for his 1945 essay in *The Atlantic*, "As We May Think," which forecast the development of machines that could help people access vast amounts of information, and his July 1945 report "Science The Endless Frontier," which provided the intellectual justification for creating the NSF.

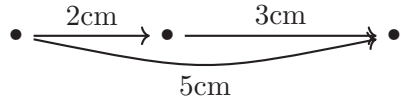
https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

uous intergraph,[10] renaming it the *differential analyzer* later that year.[11] Within a few years versions of the machine had been built and impressed into service in both the US and England. For example, differential analyzers were constructed at the Ballistic Research Laboratory in Maryland and in the basement of the Moore School of Electrical Engineering at the University of Pennsylvania, where the machines were used to compute artillery tables.[12]

3.3 Numeric Coding

Analog mechanical calculating devices like the differential analyzer (and like slide rules) take a fundamentally different approach to solving numeric equations than the digital calculators, desktop computers, laptops and cell phones with which readers of this book probably grew up. Analog machines use physical quantities like distance, speed, and the accumulation of electronic charge to directly represent numeric quantities. This approach is simplistic and straightforward, but it has many disadvantages.

For example, you can use a ruler, a pencil, and a piece of paper to add together the numbers 2 and 3: just draw a line on the paper that is 2 cm long, draw a second, connecting line that is 3 cm long, and measure the length of the resulting line:



This is the basic principle behind the slide rule, except the rules on a slide rule are drawn using a logarithmic scale, so that adding the distances results in multiplication and subtracting them results in division (Figure 3.1).

The fundamental problem with analog mechanical calculating devices is that they are limited in *precision*, the ability to distinguish two numbers; *accuracy*, the difference between the true number and the one obtained by the calculation; and *repeatability*, whether the

[10] Bush, Gage, and Stewart, "A Continuous Integraph" (1927).

[11] Bush, "The Differential Analyzer. A New Machine for Solving Differential Equations" (1931).

[12] Bunch, *The History of Science and Technology: A Browser's Guide to The Great Discoveries, Inventions, and The People Who Made Them, From The Dawn of Time to Today* (2004), p. 535.

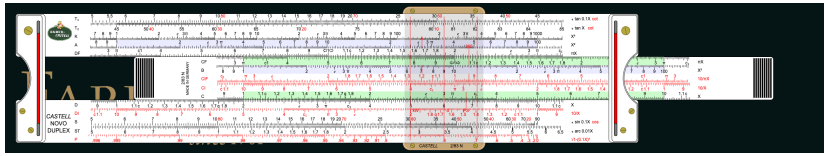


Figure 3.1. Using a slide rule to compute 2×3 = 6. The value of 2 is specified because the 1 on the C scale lines up with the 2 on the D scale. The cursor is then moved so that its center hairline is aligned with the 3 on the C scale, and the value of 6 on the D scale is the product of 2 and 3. Notice that the slide rule is simultaneously displaying that $2 \times 4 = 8$, $2 \times 5 = 10$, and many other values. it is you, the observer, who is actually doing the computation. (Slide rule simulation from www.sliderules.org/.)

same answer is obtained when following the same sequence of operations. These concepts are described in the sidebar "Precision, Accuracy, and Repeatability" on page 34.

Digital calculating systems use specific symbols – *digits* – to represent numbers and then perform math symbolically using these symbols. The mechanical computers developed by Charles Babbage in the nineteenth century used the position of wheels, rods and levers to represent decimal digits; modern computers use electric charge on a wire. Digital systems overcome many of the repeatability problems that plague analog systems by forcing intermediate physical measurements to a specific digit and then re-generating the signal. As a result, small variations in computations that result from wear or manufacturing defects can be detected and eliminated.

For example, an electronic circuit might store 5 volts (5 V) in an electronic storage device called a *capacitor* to represent a `1`, and 0 V to represent a `0`. A short while later the circuit might try to read the value: if it reads a 5 V, that's a `1`. But if a large amount of time has passed and some of the electricity has leaked out, the circuit may only read 4 V or even 3 V. As long as more than 2.5 V is read, the circuit still treats the value read as a `1`. As part of the reading operation, the circuit can then "top off" the electricity in the capacitor back to 5 V. On the other hand, if the circuit read 0.5 V, it would treat that as a `0` and not top it off—instead, it would drain the capacitor down to 0 V.

This forced choice between two values is called *digital discipline*, and it is the basis of how dynamic memory inside a modern computer works: a typical dynamic memory chip in 2020 might have 2 billion to

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

64 billion individual bits,[13] each one read and refreshed many times every second.

Like the differential analyzer, the first digital computing devices in the US were created to solve equations for scientific and military applications. The first was the Atanasoff Berry Computer (ABC),[14] built at Iowa State University by physics professor Dr. John Vincent Atanasoff and his graduate student Clifford Berry. Designed to solve systems of linear equations,[15] the ABC stored data on a pair of drums that rotated once a second. Each drum could store 32 sets of 50-bit binary numbers in 1600 capacitors: using binary numbers made the arithmetic circuits easy to design and construct. Although the basic system was functional, the input and output systems were not completed before Atanasoff was assigned by the War Department to the Naval Ordnance Laboratory in Washington, DC in September 1942. The ABC was eventually disassembled.

The second digital computing system in the US was built at the University of Pennsylvania's Moore School of Engineering by John Mauchly and J. Presper Eckert. Mauchly met Atanasoff at a scientific meeting in December 1940 where Mauchly was demonstrating an analog computer. Mauchly became interested in the promise of digital computation, and ended up traveling to Iowa and staying with Atanasoff for four days. In August 1942, Mauchly wrote a memo entitled "The Use of High-Speed Vacuum Tube Devices for Calculating," which proposed creating a fully electronic computing machine that could perform an estimated 1000 multiplications per second. The following year Mauchley was hired by Eckert, a profes-

[13]The word *bit* is short for "binary digit." Bits are the small unit of information. In normal usage we say that a bit can be either a `0` or a `1`, but they could just as well be a black or a white, or an empty or a filled. Claude E. Shannon (1916–2001), the "father" of information theory, provided a mathematical definition for the *bit* in 1948, and attributed the coinage of the word to the American mathematician John Tukey (1915–2000), although the word was in use before that time. See Garfinkel and Grunspan, *The Computer Book* (2018). They're sort of like the Greek conception of atoms, but for information. The only problem with this analogy is that in the twentieth century we learned how to split atoms; bits, in contrast, cannot be split.

[14]Using the nomenclature adopted in this chapter, the ABC is not a computer because it is not Turing Complete, a concept that we explain later in this chapter.

[15]A system of linear questions describes one or more lines in two-dimensional space, planes in three-dimensional space, or hyperplanes in multi-dimensional space. Solving the set of equations finds the place where the lines or planes intersect. Rate/time problems from first-year algebra are examples of such problems.

sor at the University of Pennsylvania, and construction started on the Electronic Numerical Integrator and Computer (ENIAC) in secret during the summer of 1943. The project was funded by the US Army's Ordnance Corps for the purpose of creating a computer that could create artillery tables, which at the time were being created nearby in Philadelphia by a group of female "computers." Several of these women, Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Meltzer, Fran Bilas, and Ruth Lichterman, became ENIAC's first programmers.

Two other early computer systems are worth mentioning. At Harvard University, Professor Howard Aiken conceived of a computer powered by relays that could perform computations and print numeric tables. Aiken partnered with IBM to design and build the computer; it was delivered to Harvard in February 1944 and started operations that summer. Called the Mark I, the machine was massive: 51 feet long, 8 feet high, and 2 feet deep. It had 500 miles of wire, 3500 relays, and 1464 10-position switches for entering numbers. Like the ENIAC the Mark I operated on decimal numbers, but because it computed with mechanical relays, rather than electronic tubes, it required 3 seconds to perform an addition and 6 for a multiplication – a thousand times slower than the machine in Philadelphia. The Mark I was built for the US Navy.

In Germany, Konrad Zuse built a series of computers: the Z1 (1936–1938), Z2 (1940), Z3 (1941), and Z4 (1945). Like the UK's Bombe and Harvard's Mark I, these computers were all built using relays. Unlike the others, none of them received significant funding from the host country's military. Zuse had to borrow money from his family and friends to construct the Z1, and he built the machine in his parents' living room! It wasn't until 1940 that Zuse received funding from the German government, and that was only partial funding. By failing to recognize the military applications of computing, the Germans squandered the significant lead in both computer theory and engineering that they had over the Allies.

3.3.1 Encoding Digital Information

Today many people tend to confuse the words *digital* and *binary*, but they are different. What makes digital computers *digital* is the use of specific, discrete values to represent information. We call these discrete values *digits*. Binary systems are digital, but they use just

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

two mutually exclusive binary digits, typically `0` and `1`. The word "bit" is actually a contraction of the words "binary" and "digit."

One of the first binary systems was the Jacquard Loom (1801), which used holes punched into wooden slats to control the pattern woven into the fabric. Each hole determined whether an individual weft would pass over or under a wrap on each pass of the shuttle through the shed. The Jacquard Loom is frequently taken as the first use of punch cards to control a piece of machinery.

It is also possible to have digital systems that use more than two values: the early ENIAC at the University of Pennsylvania (1943) used a voltage moving down 1 of 10 wires to represent the digits 0 through 9, while today's multi-level cell (MLC) flash memory uses four discrete voltage levels within each flash cell, allowing them to store two bits per cell.[16] Not surprisingly, MLC flash costs less than single-level cell (SLC) flash memory, but it is more prone to errors.[17]

Digital computers need a way to store information and to read back the information that they have stored. The Jacquard Loom wasn't a computer because it had no way of writing to its punch cards: the same was true of the card sorters and tabulators that Herman Hollerith created for the 1890 US Census. Without such memory, these devices lacked the ability to alter computations based on an earlier calculation, thus failing the definition for computing. In contrast, the flash memory (1980) in a modern cell phone can be both read and written.

Computers can store all kinds of information beyond simple binary bits: even in the 1940s, computers were computing on integers,

[16] High-dimension storage and communication are active research areas in quantum technology. Some are investigating qutrits, quantum bits that have three states. Separately, one group has demonstrated that it can use modulators and mirrors to encode information in photons along seven dimensions, exploiting the photon's "orbital angular momentum" and "angular position" instead of polarization, which is the typical approach. See Mirhosseini et al., "High-Dimensional Quantum Cryptography with Twisted Light" (2015).

[17] Analog computers, in contrast, might use a specific voltage to represent the value of 1, half that voltage to represent the value of 0.5, twice that voltage to represent the value of 2, and so on. Although you might think that this approach provides for more flexibility, the problem is that there is no good way for such computers to distinguish values that are close together, like 1.001 and 1.002. As a result, analog computers tend to lack both accuracy and repeatability, as discussed in the sidebar "Precision, Accuracy, and Repeatability" on page 34. This is also the fundamental problem of proposals to use analog computers as an alternative to quantum computers.

floating point numbers, and text. Today's computers can store virtually any kind of information that can be contemplated, including pictures, sound, and movies.[18] Fundamentally, all of these things are ultimately transformed into a series of bits and recorded in the computer's memory, and then reconstructed on output.

Representation is a word that computer scientists use to describe how information is broken down and stored. One of the simplest representations uses different combinations of binary digits to represent different integers. For example, if you have three binary digits, you can represent eight different values, typically taken to be the numbers 0 through 7:

Bits			Value
A	B	C	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

In modern computers, data is arranged in groups of eight bits called *bytes*. A byte can represent $2\times2\times2\times2\times2\times2\times2\times2 = 2^8 = 256$ different values. This is typically scaled from 0 to 255, but it can also be scaled from −128 to 127. The first case is sometimes called an *unsigned 8-bit integer*, the second a *signed 8-bit integer.*

It is common to group four bytes together to form a 32-bit *word* that can represent numbers from −2 147 483 648 to 2 147 483 647. Rational numbers can be represented with two numbers, one for the numerator, one for the denominator. Alternatively, there are *floating point* representations; the IEEE single-precision floating point format uses 32 bits to represent floating point numbers: 1 bit for the number's sign, 8 bits for the exponent, and 23 bits as a binary

[18] Some things that modern computers can't store are complex physical objects, thoughts, space, time, or entanglement states.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

fraction.[19] However, today most computations are done with 64-bit IEEE *double-precision* floating point numbers, since the additional four bytes of storage is typically inconsequential while the increase in precision is dramatic.

Computers also use combinations of bits to represent individual letters, like the letters typed into a computer that eventually became the sentence you are reading. Using combinations of bits to represent letters dates back to 1874, when the French inventor Émile Baudot devised a more effective way to send text down a telegraph line. Instead of using the dots and dashes of Morse code, Baudot designed a device with five keys and a rotating "distributor" that electronically connected a switch at the end of each key, in rapid succession, to the line. The device sent down the telegraph line a rapid succession of electric pulses corresponding to whether each key was up or down. Today this approach is called *time-division multiplexing*. Five bits allowed the operator to send one of 32 possible combinations down the line with each rotation of the distributor. Baudot used 27 of these codes for letters (E and É were represented with different codes) and another two for the space character and a marker for the end of the message. A device at the other end recorded the marks on paper: it didn't take long to invent devices that actually printed letters that corresponded to the codes that the operator was sending. And thus was born the printing telegraph, soon to be known as the teletype.

Just as the way that numbers are stored inside computers has been standardized, so too has the way that letters are stored. In the 1960s much of the industry adopted the American Standard Code for Information Interchange – ASCII – which dictates that letter "A" will have the binary code `0100001`, the letter "B" will be `0100010`, "C" will be `0100011`, and so on. Lower case letters start with "a" at `0110001`. These numbers correspond to the values 65, 66, 67 and 97 in decimal (base 10). In the 1990s ASCII was expanded to include the complex glyphs of Japanese, Chinese, Korean and all of the world's other languages. The new system is called UNICODE and has since been expanded to include dead languages like Cuneiform and even made-up languages like Klingon.

[19] Because numbers like 0.1 cannot be perfectly represented as a binary fraction, when floating point numbers like 0.1 are repeatedly added together, the result might end up as 0.9999999999999999 instead of 1.0. This is called *roundoff error*.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

3.3.2 Digital Computation

Computers need to have a way to change their behavior based on the information that they read – that is, they need a way to *compute*. Computer engineers use the term *logic* to describe both the internal rules that a computer follows and the mechanism that implements those rules. Once again, logic can be built from many different technologies: from the point of view of a *computer scientist* the details of how the logic is actually implemented doesn't matter much.[20] In an electronic computer, the logic is assembled from fundamental building blocks called *gates*.

Gates can have 1 or more inputs and 1 or more outputs. These inputs and outputs are typically wires, but in a diagram you will see them drawn as lines that carry digital information. The simplest gates replicate the basic logic operations of Boolean algebra:

- The AND gate (Figure 3.2) combines its inputs and produces a 1 if both of its inputs are 1, otherwise its output is 0.
- The NOT gate (Figure 3.3) has an output that is the reverse of its input.

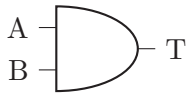
Any logic circuit can be created using combinations of just these two gates and the appropriate connecting wires.

For example, you can make an AND gate that has three inputs (A, B, and C) by taking the output of a single AND gate that computes (A AND B) and connecting it along with C to the input of a second AND gate, creating a circuit that computes ((A AND B) AND C). More generally, it is possible to use AND and NOT gates to build complex circuits that add, subtract, multiply, or divide numbers. For example, Figure 3.4 shows how such circuits are put together to create a one-bit "full-adder," while Figure 3.5 shows how four full-adders can be combined to form a four-bit adder.

It is also possible to create circuits that interface with memory units to load and store information. It is even possible to use a combination of AND and NOT gates to create memory units – such memory is called *static memory* and is much faster than other kinds

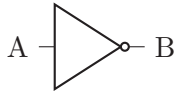
[20] In the 1970s, Danny Hillis and Mitch Kapor (who later went on to found the Lotus Development Corporation) created a computer out of Tinkertoy that played Tic Tac Toe. The computer is now part of the permanent collection at The Computer History Museum. See D. Hillis and Silverman, "Original Tinkertoy Computer" (1978).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press



inputs		outputs
A	B	T (A AND B)
False	False	False
False	True	False
True	False	False
True	True	True

Figure 3.2. A simple AND gate and its "truth-table."



input	output
A	B
False	True
True	False

Figure 3.3. A simple NOT gate and its truth table.

of memory used inside a computer. In fact, *any* digital circuit can be built if you can combine sufficient numbers of AND and NOT gates with the correct wiring pattern. For this reason, the combination of these gates is said to be *universal*.

But one can do even better: the AND and the NOT gate can be combined into a single universal gate called the NAND – *not AND* – gate, from which *every* digital circuit can be built.

In practice, digital designers use all kinds of gates, safe in the knowledge that their designs can always be transformed in a series of universal NAND gates if needed. In fact, the process for doing this is so straightforward and automatic that such transformations can happen when a design is turned into silicon without the designer even knowing it.

3.4 Computing, Computability and Turing Complete

There are many questions to ask in comparing these computers and trying to assess the role that they played in World War II. What sort of monetary and human resources were required to build each machine? How hard was it to find skilled scientists to work on these projects? How much original research had to be done? Did these devices actually contribute to the war effort, as the machines at Bletchley Park clearly did, or were they merely fascinating historical footnotes, like the Zuse machines? One might consider their contribution to military efforts after the war: ENIAC's first official calcu-

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

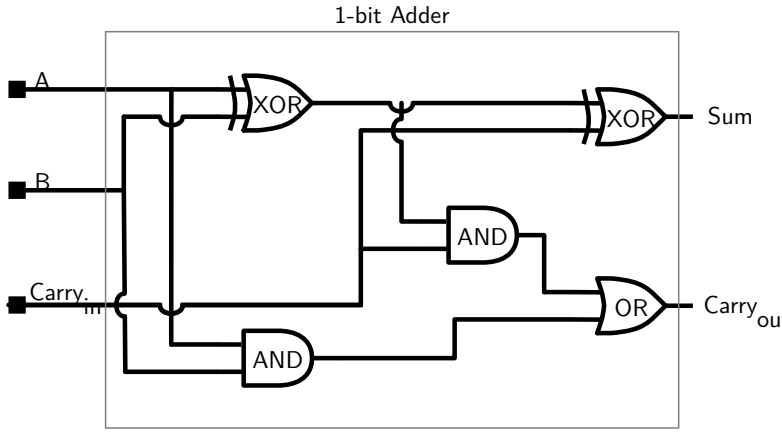


Figure 3.4. Circuit diagram of a "full adder." The inputs are A, B, and C (carry). The outputs is S (the sum) and Cout (carry out). S is true if either A, B, or C are true. If two of them are true then Cout is true and S is false. If all three inputs are true, then both S and Cout are true. Multiple full adders can be chained together to add any number of binary bits.

$$A_3A_2A_1A_0 + B_3B_2B_1B_0 = S_3S_2S_1S_0$$

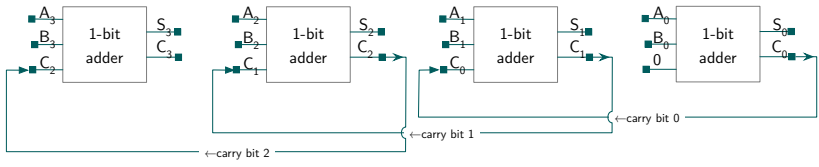


Figure 3.5. Four one-bit full-adders can be combined to form a four-bit adder. Each bit adds the input bits A_n and B_n and the carry bit C_{n-1}. (Note: This four-bit adder ignores the carry bit C_3. As a result, adding `1111` and `0001` will produce `0000`, a condition known as an overflow.) This circuit is "clock-free," meaning that it runs without reference to an external clock, although it may take a few hundred picoseconds for the transistors that make up the gates to stabilize when the logic inputs change. Compare this with Figure 6.3, a 4-bit quantum adder.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

lations were not for artillery tables, but for the development of the hydrogen bomb, and the team went on to create the Universal Automatic Computer (UNIVAC), while the UK's obsession with secrecy and its persecution of Alan Turing were major setbacks for the early UK computer industry.

Computer scientists evaluating these early machines tend to focus on two questions: how fast could the machines calculate and were they *Turing Complete*?

Speed. For the pioneers of the 1940s, faster calculations were the *only* reason that justified spending the time and money that it took to create calculating machines. It was clear that mechanical calculation had a much higher *initial cost* than human computers but a much lower *incremental cost*. Within the world of mechanical computation, electromechanical systems built with relays had a lower initial cost than electronic systems built with tubes, as the technology was better understood and more readily available. It was also a thousand times slower.

Turing Complete. Modern computers are said to be *general purpose machines*, in that they can be programmed to perform any calculation or any programmable function. This is sometimes called *Turing Completeness*, meaning that the computer implements the computational model described by Alan Turing.[21] Being Turing Complete is what differentiates a machine that calculates from one that computes. The easiest way to make a machine that is Turing Complete is to have it store the program in some kind of memory and for there to be some way to change the program's order of execution, either a mechanism that allows the program to modify itself, or to have the program's execution determined by a computed data value.

[21] Turing developed his model to solve a challenge posed by the mathematicians David Hilbert and Wilhelm Ackermann in 1928 called the Entscheidungsproblem (German for "decision problem"). The problem was to develop a procedure or algorithm for evaluating any mathematical statement to determine if it is true or false. Turing developed his model of computation to show that this was not possible; the American mathematician Alonzo Church also showed the impossibility of the Entscheidungsproblem, although using a completely different approach. Church published his solution (Church, "An Unsolvable Problem of Elementary Number Theory" (1936)) a few months before Turing (Turing, "On Computable Numbers, with an Application to The Entscheidungsproblem" (1936)); today these solutions are called the Church–Turing thesis or the Church–Turing hypothesis.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

Surprisingly by today's standards, the pioneers were not concerned with storage the way we are today. Modern computers have storage systems that can both store data and load it back: such storage is used for both programs and data. But storage that could support such "load and store" operations on the early computers was minuscule. Code-breaking the ENIGMA required rapidly iterating through many possible encryption keys, but the intermediate results did not have to be archived. Cracking each Lorenz cipher required a lot of input data, which was provided on paper tape, but there was very little in the way of output. Creating artillery tables required a computer-controlled teleprinter, but such devices were write-once, read-never. Moreover, such printers were widely available in the 1940s, as they had been developed for printing telegraphs in the early 1900s.[22]

After the war, the pioneers turned their attention to building machines that could be easily reprogrammed to different tasks. This created the need for some sort of system that could be used to store the programs. Three main technologies emerged: first, acoustic delay lines, in which bits were stored as pulses of sound traveling down a tube of mercury (although Alan Turing suggested using gin instead); second, drum memory, in which bits were stored by changing the magnetization of a small region of a rotating magnetic drum; third, core memory, in which bits were stored by changing the magnetization of a tiny iron torus. Of these three, magnetic core became the dominant form of memory until the emergence of semiconductor memory in the late 1960s, and was widely used until the late 1970s.

3.4.1 Introducing The Halting Problem

In 1936, Alan Turing invented a modern concept of computers when he proved that it is impossible to examine a computer program and determine if the program will halt or will run forever. Here we present Turing's idea by showing that such a program-analyzing program must sometimes be wrong. This is called a proof by *contradiction*.

[22]The Morkrum Company, established in 1906 by Charles Krum and the Morton family, developed the M10 printer in 1908. It was adopted by the Associated Press in 1915. The company merged with the Kleinschmidt Companies in 1925, and in 1929 the combined company changed its name to Teletype after the name of its most successful product. See D. R. House, "A Synopsis of Teletype Corporation History" (2001).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

There are some programs that obviously halt:

```
PROGRAM A:
 1: PRINT "Hello World."
 2: HALT
```

Thus, `HALT_CHECK(PROGRAM A) = HALTS`.

Likewise, there are programs that obviously do not halt:

```
PROGRAM B:
 1: PRINT "Hello World."
 2: GOTO 1
```

Thus, `HALT_CHECK(PROGRAM B) = DOES NOT HALT`.

Here we use *functional notation* to denote a computer program called `HALT_CHECK` that examines a second computer program (variously `PROGRAM A` and `PROGRAM B`) and returns `HALTS` or `DOES NOT HALT`.

If only a program like `HALT_CHECK` could exist! With it, we could answer *any* mathematical question! For example, we could use it to determine the correctness of Fermat's Last Theorem, which holds that there is no solution to equation $A^n + B^n = C^n$ for $A > 0$, $B > 0$, $C > 0$ and $n > 2$. We would just code up a new program called `FERMAT`:

```
PROGRAM FERMAT:
 1: A ← 1
 2:   B ← 1
 3:     C ← 1
 4:       N ← 1
 5:         IF A^N + B^N = C^N THEN
              PRINT "FERMAT'S LAST THEOREM DISPROVED!"
              PRINT A,B,C,D
              HALT
            N ← N + 1
 6:         IF N < C THEN GOTO 5
         C ← C + 1
 7:       IF C < B THEN GOTO 4
       B ← B + 1
 8:     IF B < A THEN GOTO 3
 9:   A ← A + 1
      GOTO 2
10: THIS LINE WILL NEVER BE REACHED
```


https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

We would then compute `HALT_CHECK(FERMAT)`. If the result was `DOES NOT HALT`, we know that `FERMAT` never halts, and thus Fermat's Last Theorem is true!

3.4.2 The Halting Problem Cannot Be Solved

Sadly, the Halting Problem cannot be solved. Computer scientists say that the function `HALT_CHECK` is *undecidable* or *uncomputable*.

To see why we cannot create a `HALT_CHECK` program that works reliably, in all cases, we simply construct a second program, which we will call `H2`:

```
PROGRAM H2:
 1: IF HALT_CHECK(H2) = HALTS, GOTO 1
 2: PRINT "H2 HALTS!"
 3: HALT
```

Program `H2` asks `HALT_CHECK` if `H2` itself halts. If `HALT_CHECK` reports that `H2` halts, then `H2` runs forever. But if `HALT_CHECK` reports that `H2` runs forever, then it must not halt, so `HALT_CHECK(H2)=False`. But then H2 halts! Clearly, `HALT_CHECK` cannot correctly report if `H2` halts or runs forever.

Program `H2` is the logical equivalent of what's called the Liars Paradox. The paradox is that when a person says "I am lying," they are speaking a contradiction. If the person is telling the truth, then they are lying. But if they are lying, then they are telling the truth. So `HALT_CHECK` can't exist, and finding out if Fermat's Last Theorem is true or not requires years of mathematical research, rather than simply coding up the question and giving it to a computer.[23]

The theory of computation is a lot of fun intellectually, and it is closely related to Gödel's theorem of incompleteness, which holds that in any system of mathematics there are statements – an infinite number, in fact – that are true but unprovable. In fact, it is possible to use the theory of computation to prove Gödel's theorem. But the core ideas of Turing's theory give us more than a simple parlor game that lets us show that some functions are not computable: it gives us a theory that allows us to prove that the only difference between

[23]The British mathematician Sir Andrew Wiles published two papers proving Fermat's Last Theorem in 1995; combined, the papers totaled 129 pages and required more than seven years of research. Wiles was knighted as a result of his accomplishment and received the Abel Prize, which is generally regarded as the Nobel Prize of mathematics.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

different computers is the size of a problem that they can process, and the speed with which they can arrive at a correct answer. That is, computability concerns *whether a computer can* perform some task, and not how long that task will take or how much memory and storage is necessary. Unfortunately, we are limited by time and memory. The time and other practical limits on computation are the domain of "complexity theory," which we discuss in Section 3.5 (p. 98).

3.4.3 Using The Halting Problem

To recap, the theory of computation tells us that even given a computer that is *infinitely powerful*, has an *infinite amount of storage*, and an *unlimited amount of time,* there are still problems that cannot be solved. The Halting Problem is one such problem.

One of the best uses that you can make of the Halting Problem is as a kind of snake oil detector. For example, upon close examination, many disreputable computer security companies are effectively claiming to have solved the Halting Problem.

Consider a (hypothetical) company that claims to have an antivirus program called `WIPE_CHECK` that can determine with perfect accuracy if a cell phone app can wipe your cell phone. If such a program existed, we could use it to solve Fermat's Last Theorem! All we would have to do is write a new program and test it with `WIPE_CHECK`:

```
PROGRAM FERMAT-WIPER:
 1: A ← 1
 2:   B ← 1
 3:     C ← 1
 4:       N ← 1
 5:         IF (A^N) + (B^N) = C^N THEN
              PRINT "FERMAT'S LAST THEOREM DISPROVED!"
              PRINT A,B,C,D
              PRINT "NOW WIPING YOUR PHONE"
              WIPE_CELL_PHONE
            N ← N + 1
 6:         IF N < C THEN GOTO 5
         C ← C + 1
 7:       IF C < B THEN GOTO 4
       B ← B + 1
 8:    IF B < A THEN GOTO 3
```


https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

```
 9:   A ← A + 1
      GOTO 2
10: THIS LINE WILL NEVER BE REACHED
```

Something here must be wrong! If a program called `WIPE_CHECK` could really examine *any* program and always, reliably, determine if that program could wipe your phone, then the program-analyzing program would need to be at least as powerful as `HALT_CHECK`, because we could use it to solve the same problems.

As with `HALT_CHECK`, we can prove that `WIPE_CHECK` cannot exist by using contradiction:

```
PROGRAM W2:
 1: IF WIPE_CHECK(W2) = WILL_WIPE_PHONE THEN GOTO 1
 2: WIPE_CELL_PHONE
```

`WIPE_CHECK(W2)` *cannot* return the correct answer, for the same reason that `HALT_CHECK(H2)` cannot: if `W2` wipes your phone, then it doesn't, but if it doesn't wipe your phone, then it does. Clearly, a perfectly accurate `WIPE_CHECK` program cannot exist.

3.5 Moore's Law, Exponential Growth, and Complexity Theory

Computing's pioneers realized that computers would get faster and that storage capacities would increase with every coming year – in principle, they realized, there is no limit to how fast computers could get or how much they could store.

For example, in his seminal 1951 article "Computing Machinery and Intelligence," Alan Turing wrote that in 50 years' time computers would have a storage capacity of 1×10^9 (1 000 000 000) binary digits. As it turned out, he was right: Apple's PowerBook G4, a laptop introduced on January 9, 2001, came with 128 MiB of memory (1 073 741 824 bits), expandable to 1GiB (8 589 934 592 bits).

In his article, Turing hypothesized that a person chatting (by text!) simultaneously with such a computer and a second person would be unable to distinguish between the computer and the second person roughly 70 percent of the time. This challenge is the infamous "Turing Test." Yet here Turing over-estimated the powers of his fellow humans: communications from Joseph Weizenbaum's ELIZA program were regularly mistaken for those of a human just a few months after it was operational in 1964, and many so-called "chatterbot" programs have passed versions of the Turing Test since

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

the 1990s. Today the Internet is awash with programs that not only imitate humans, but attempt to get them to take actions in the physical world, all without revealing that they are bots. And even when users know they are interacting with software, some treat them as people, fall in love with these computer personalities, and take major life decisions based on interactions with them.[24]

Turing's predictive powers were pretty amazing when you consider that the computer Turing built in 1950 – the Pilot ACE (Automatic Computing Engine) – had a main memory of just 4096 bits (arranged as 128 32-bit words). Turing was predicting that the storage capacity of computers would increase by a factor of a 250 000 in 50 years. He pretty much nailed it.

Other engineered systems have not enjoyed similar continued growth in speedup. Consider the passenger airplane:

- In 1903 the Wright Flyer reached an airspeed of 31 mph. It carried one person.
- In 1957 the Boeing 707-020 jet aircraft had a cruising speed of 600 mph;[25] it carried 140 passengers.
- Between 1976 and 2003, the Concorde supersonic jet ferried well-heeled passengers across the Atlantic at 1340 mph. The Concorde carried 92 to 128 passengers.
- The Boeing 787 Dreamliner made its debut 2011, with a maximum operating speed of 600 mph and a cruising speed of 560 mph. The Dreamliner carries 242 passengers.

Planes have certainly improved over the past 100 years. They can carry more passengers and do so more safely. But no technical metric over the past 100 years, from fuel efficiency to safety to cost, compares to the performance improvements that computers have experienced in just 50. Computers have experienced eye-popping increases in speed of computation, storage – and in the efficiency of their algorithms.

In part, planes are limited by the physics of sound: the speed of sound where jets fly is roughly 660 knots, and planes experience

[24] Olson, "My Girlfriend Is a Chatbot" (2020).

[25] Repantis, "Why Hasn't Commercial Air Travel Gotten Any Faster Since The 1960s?" (2014).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

significant turbulence as they approach it, thus creating a real "barrier" that planes must be engineered to overcome. No similar barrier exists in the world of computing. Planes must overcome the physics of moving large objects: computers need move only electrons.

Turing's Pilot ACE computed with 800 vacuum tubes, but within a few years computers were being constructed with semiconductor transistors. In 1965 Gordon Moore, who at the time was director of research and development at Fairchild Semiconductor, wrote an article exploring the technology trends that the semiconductor industry was facing. Unlike aircraft, semiconductors are not made one at a time: they are made in batches on round disks of silicon called *wafers* and then cut up into individual chips and put into packages that we think of as integrated circuits:

> At present, packaging costs so far exceed the cost of the semiconductor structure itself that there is no incentive to improve yields, but they can be raised as high as is economically justified. No barrier exists comparable to the thermodynamic equilibrium considerations that often limit yields in chemical reactions; it is not even necessary to do any fundamental research or to replace present processes. Only the engineering effort is needed.[26]

What this meant, Moore wrote, is that the number of components on semiconductors was likely to rise exponentially over time "at a rate of roughly a factor of two per year." He added: "certainly over the short term this rate can be expected to continue, if not to increase." Eventually this prediction was named *Moore's Law* and the rate was scaled back to a doubling every 18 months.[27]

The increase in computing over the past 50 years has truly been incredible. In the 1940s the ENIAC could perform 350 multiplications per second; today one can purchase a high-end graphical coprocessing card for under $6000 that can perform "100 teraflops," or 10^{14} floating point operations per second, an increase of roughly 3×10^{11}.[28] Iowa State's ABC stored 3200 bits in the size of an actual

[26] Moore, "Cramming More Components Onto Integrated Circuits" (1965).

[27] Moore, "Progress in Digital Integrated Electronics [Technical Literature, Copyright 1975 IEEE. Reprinted, with Permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11–13.]" (2006).

[28] The "floating point" operations referred to in the measure "flops" are typically addition, subtraction, multiplication, or a multiplication paired with an addition.

desktop; today you can purchase a desktop disk array with six 16 TB drives for under $6000 that stores roughly 8×10^{14} bits, an increase of roughly 2×10^{11}.

Danny Hillis (b. 1956) is a beloved, accomplished, insightful computer scientist and innovator. He earned his Ph.D. at MIT (advised by Marvin Minksy and Claude Shannon), founded the supercomputer company Thinking Machines in the 1980s, and went on to be a Fellow at the Walt Disney Company. Hillis once gave a talk at the New York City Hilton in which he predicted that one day computers would be so inexpensive that they would be everywhere – in numbers exceeding the world's population. "What are you going to do with all of them?" a heckler in the audience shouted. "It's not as if you want one in every doorknob."

In the 1990s, Hillis returned to the hotel and noticed that each door had been equipped with an electronic lock. "You know what?" he told the audience at the tenth anniversary of the MIT Media Lab. "There is a computer in every doorknob!"[29]

Moore's Law held until roughly 2016, when the market leader in chip production, Intel, signaled that developments in chip-shrinking would slow.[30] In part, this was a reflection of economic realities: for many years Intel and other companies had moderated their technology investments to match the prediction of Moore's Law, bringing a breath of predictability to the topsy-turvy world of high-tech. But starting in the 2000s, other factors such as power consumption came to dominate semiconductor design requirements: no reasonable amount of technology investment could keep Intel on the technology curve that had been forecast in the 1960s. This slowdown was also a result of quantum effects – as gate sizes shrink, there's a greater chance for electrons to "tunnel" from one semiconductor tract to another, causing an error.

Moore's Law isn't really a law: it's really a prediction about the likely progress in semiconductors, given continued investment of dollars in research, engineering and production.

But it is not a precise measurement, because any given processor typically takes a different amount of time for each of these operations, and the amount of time that it takes can also depend on the input data. The ENIAC did not support floating point operations, but most of its contemporary systems did.

[29] Garfinkel, "1985–1995: Digital Decade. MIT's Computing Think Tank Chronicles The Electronic Age" (1995).

[30] Simonite, "Intel Puts The Brakes on Moore's Law" (2016).

3.5.1 Software Speedups

Computers operate from the interplay of hardware and software. In the last section we recounted the dramatic improvements in storage capacity and speed that hardware has experienced over the past 50 years. There have also been improvements in software, but those improvements are of a fundamentally different nature and harder to quantify. This is relevant for our exploration of quantum computing, as the performance that quantum promises is paradoxically much closer to performance improvements of the kind that software has experienced.

Software performance improvements are primarily the result of improvements in algorithms and data structures. An *algorithm* is a method, typically described by a sequence of steps, that performs some kind of computation. *Data structures* refer to the stylized ways that information is stored inside a computer's memory.

It is difficult to quantify changes to an algorithm or a data structure that can change the performance of a system, because performance depends on a dizzying number of specifics.

For example, consider a simple database of the first 17 US presidents (Table 3.1). Each president's information is stored in a *record*, and each record is put in a row, which are numbered 0–16. The records are sorted by the president's date of birth (normalized to the Gregorian calendar). This database is a data structure. Let's say that the computer's memory in which this data structure is stored allows *random access* – that is, it can immediately access any record by simply knowing the row number.

Now, let's say that we need two algorithms. The first is called `BIRTHDATE_TO_PRESIDENT`; given a president's birthdate, it returns the president's name. A simple algorithm would be:

```
ALGORITHM BIRTHDATE_TO_PRESIDENT(DATE):
BEGIN VARIABLES
N: 0
ROW: DATABASE Table 3.1

BEGIN CODE
 1: IF ROW[N].birthday = DATE:
      PRINT ROW[N]
      HALT
 2: N ← N + 1
 3: IF N < ROW.length:
```


https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

```
     GOTO 1
4: PRINT "NO PRESIDENT WITH BIRTHDAY", DATE
5: HALT
```

Here we have introduced some notation. `N` is a variable that can hold any number. `N` is initialized to zero when the program is loaded. `ROW` is an array of records drawn from Table 3.1. `ROW[0]` is the first row in the database, and `ROW.length` is the total number of rows, in this case the number 17. The program starts by checking to see if the row referenced in the database has a `date_of_birth` equal to the `DATE` that is provided when the program starts running. If it does, the program prints the entire record and stops. If the record at `ROW[N]` does not have the requested birthdate, line 3 increments the value of N by 1. Line 3 causes the algorithm to jump back to line 1 if the N is less than the number of rows (17). If N is 17 then line 4 runs: the program prints that there is no president with that birthdate and stops.

The amount of time this program takes to run[31] depends on many factors, such as:

1. The amount of time it takes to load the program into memory and start execution.
2. The amount of time it takes to set variable N to zero.
3. The amount of time it takes to fetch the contents of ROW[N].
4. The amount of time it takes to compare two dates.
5. The amount of time it takes to increment N.
6. The amount of time it takes to compare N to the number 17.
7. The amount of time it takes to jump from line 3 to line 1.
8. Whether `DATE` is in the database or not.

Times 1 and 2 are constant for any database. Times 3 through 7 are the amount of time that it takes to check any given record. If

[31] When examining algorithms like this, it is common for computer scientists to consider both average and worst-case performance. In this example we only consider worst-case performance.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

Table 3.1. The first 17 US presidents, sorted by date of birth.

row	Birthday	President	#	Tenure
[0]	1732-02-22	George Washington	1	Apr 30, 1789 – Mar 4, 1797
[1]	1735-10-30	John Adams	2	Mar 4, 1797 – Mar 4, 1801
[2]	1743-04-13	Thomas Jefferson	3	Mar 4, 1801 – Mar 4, 1809
[3]	1751-03-16	James Madison	4	Mar 4, 1809 – Mar 4, 1817
[4]	1758-04-28	James Monroe	5	Mar 4, 1817 – Mar 4, 1825
[5]	1767-03-15	Andrew Jackson	7	Mar 4, 1829 – Mar 4, 1837
[6]	1767-07-11	John Quincy Adams	6	Mar 4, 1825 – Mar 4, 1829
[7]	1773-02-09	William Harrison	9	Mar 4, 1841 – Apr 4, 1841
[8]	1782-12-05	Martin Van Buren	8	Mar 4, 1837 – Mar 4, 1841
[9]	1784-11-24	Zachary Taylor	12	Mar 4, 1849 – Jul 9, 1850
[10]	1790-03-29	John Tyler	10	Apr 4, 1841 – Mar 4, 1845
[11]	1791-04-23	James Buchanan	15	Mar 4, 1857 – Mar 4, 1861
[12]	1795-11-02	James K. Polk	11	Mar 4, 1845 – Mar 4, 1849
[13]	1800-01-07	Millard Fillmore	13	Jul 9, 1850 – Mar 4, 1853
[14]	1804-11-23	Franklin Pierce	14	Mar 4, 1853 – Mar 4, 1857
[15]	1808-12-29	Andrew Johnson	17	Apr 15, 1865 – Mar 4, 1869
[16]	1809-02-12	Abraham Lincoln	16	Mar 4, 1861 – Apr 15, 1865

`DATE` is not in the database, then the total amount of time will be proportional to the sum of times 3 through 7.

Because the birthdates are sorted, we could try to improve the algorithm by having it stop when DATE is larger than the date of birth of the president in `ROW[N]`:

```
ALGORITHM BIRTHDATE_TO_PRESIDENT2(DATE):
BEGIN VARIABLES
N: 0
ROW: DATABASE Table 3.1

BEGIN CODE
 1: IF ROW[N].birthday = DATE:
      PRINT ROW[N]
      HALT
 2: N ← N + 1
 3: IF (N < ROW.length) AND (ROW[N].birthday <= DATE):
    GOTO 1
 4: PRINT "NO PRESIDENT WITH BIRTHDAY", DATE
 5: HALT
```

Unfortunately, it isn't immediately clear if this change actually improves the performance of the algorithm. If the date being re-

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

quested is somewhere before the end of the list, the algorithm will stop early, but if the date requested is after February 12, 1809, the algorithm will still need to scan the entire database. And as an added penalty, there are now two comparisons on line 1 each time the comparison each time through the loop.

A better approach is to use what's known as a binary search:

```
ALGORITHM BIRTHDATE_TO_PRESIDENT_BINARY_SEARCH(DATE):
BEGIN VARIABLES
GUESS: 0
MIN: 0
MAX: 16
ROW: DATABASE Table 3.1

BEGIN CODE
 1: IF MAX < MIN:
      PRINT "DATE NOT FOUND"
      HALT
 2: GUESS ← INTEGER (( MIN + MAX ) / 2 )
 3: IF ROW[GUESS].birthday is DATE:
      PRINT ROW[GUESS]
      HALT
 4: IF ROW[GUESS].birthday < DATE:
      MIN ← GUESS + 1
      GOTO 2
 5: MAX ← GUESS - 1
 6: GOTO 2
```

This program is more complicated than the first, but in the worst case it only needs to check 5 of the rows, not 17. Mathematically, we can say that its typical performance is going to be proportional to the base-2 logarithm of the size of the table, rather than the length of the table.

Computer scientists have a notation for describing this performance concept succinctly called *Big-O notation*. Using this notation, we can describe the runtime of the first two algorithms as $O(n)$ because the runtime is proportional to the length of the table (n), while the third algorithm has a runtime of $O(\log n)$ because its runtime is proportional to the natural log.

As a final thought, all of the examples in this section assume that the records in the database were stored in sorted order. If they aren't

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

stored in some knowable order, then the only search that works is a sequential search from the beginning to the end. In a real application we would want to be able to search by not just birthdate, but by the other fields as well. A modern database management system would handle this by having additional tables called *index tables*, one sorted by name, one sorted by birthdate, and so on. These tables would consist of just the item being indexed and the row number.

3.5.2 Polynomial Complexity (P)

Programs to sort and search through databases were among the first to be written by computing's pioneers. John von Neumann's first computer program for the Electronic Discrete Variable Automatic Computer (EDVAC, the successor to the ENIAC) was a program to sort numbers, and von Neumann concluded that the EDVAC would be "definitely faster" at sorting than special purpose hardware that IBM had created for sorting punch cards, which could sort about 400 cards/minute.[32] Then von Neumann realized that he could improve the speed of his program by a factor of 80 simply by making changes to the EDVAC's hardware and corresponding changes to the program.

Early computer systems were extremely limited in their main memory, so sorting programs had to perform complex sequences in which data was read from one tape and written to others. A surviving article by Remington Rand describes how to sort data on its UNIVAC computer with six tape drives, and notes that it is possible to sort 12 000 10-word items (a full tape) in just 28 minutes.[33,34]

If all of the numbers to be sorted can fit into a computer's memory, the most obvious way to sort is something called an *exchange sort* or *bubble sort*. The algorithm is simple: start at the beginning of the list and see if the first two numbers are out of order. Now consider the second and third numbers, swapping them if they are out of order. Continue to the end of the list, then start again at the beginning. Repeat until the list is sorted. This approach never fails to produce a sorted list, but it requires n passes through the list to assure completion, where n is the number of elements in the list.

[32] Knuth, "Von Neumann's First Computer Program" (1970).

[33] Remington Rand, *Sorting Methods for UNIVAC Systems* (1954).

[34] The UNIVAC had a word size of 72 bits. For comparison, a 3GHz Intel Core i5 microprocessor can sort on array of 12 000 floating point values, each of which requires 192 bits, in 4.5 milliseconds.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

Since each pass through also requires $n-1$ comparison and swap operations, the algorithm requires at most $(n)(n-1) = n^2 - n$ operations. As n gets large the value n^2 dominates the value $(-n)$, so we say that bubble sort requires "order n squared" time to solve, which is written $O(n^2)$. It is said that bubble sort "requires polynomial time" or that it has "polynomial complexity." Here, the polynomial is n^2.

There are a few obvious ways to improve on the bubble sort algorithm presented above, but it is hard to improve it by more than a factor of two. Then in 1959, Donald Shell came up with a fundamentally new sort algorithm that is now called Shell Sort. Although Shell Sort still has $O(n^2)$ performance in the worst case, it typically runs much faster. Two years later, Tony Hoare invented one of the best sorting algorithms we have today, known as Quicksort. It also has $O(n^2)$ worst-case performance, but its average performance is $O(n \log n)$.

All of these sort algorithms have performance in P, because they all take an amount of time to sort the array that is proportional to a geometric function of the function of the array's length. But in real world situations, some of these algorithms are faster than others. When sorting large datasets, such performance improvements can be dramatic.

3.5.3 Nondeterminism

Sorting turned out to be one of the easier problems for the pioneers to conquer: a harder one was scheduling, such as the classic traveling salesperson problem (TSP). Here we provide a simple variation of the problem:

> A sales representative needs to visit 20 cities by car and can only drive 350 miles on a single tank of gas: is it possible to reach all 20 cities in a single day without refueling?

If any two of the cities are more than 350 miles apart, then the answer is obviously no. But if the cities are scattered throughout Pennsylvania (which is 285 miles across), and some of the cities are directly connected by roads while others aren't, the answer to the question isn't obvious. If all of the cities are within a mile of the main branch of the Pennsylvania Turnpike, then the answer is clearly yes. But what if some of the cities are close to the Turnpike's Northeast

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

Extension? What if one of the cities is State College (not quite a city, but home of Penn State University), and far from both the Turnpike and the Northeast Extension?

With 20 cities there are actually $20 \times 19 \times 18 \ldots 2 \times 1 = 20! = 2.43 \times 10^{18}$ different ways of driving between them in theory, which is *way* too many to consider with even a modern computer.[35]

Complexity Theory, which is a part of *Theoretical Computer Science*, is the branch of computer science that is devoted to understanding the differences between problems like sorting and the TSP.[36] *Operations research* is the academic discipline that has taken on solving problems like this. Operations research emerged as a field during the Second World War for solving problems such as shipping supplies, deciding how much armor to put on aircraft, and searching for submarines. Problems like TSP arise on a daily basis for organizations that are trying to make optimal use of their fuel and vehicles. Today airlines and delivery companies solve versions of these problems when trying to decide where they should buy fuel and the routes that their vehicles should travel.

This version of TSP is called a decision problem: the answer is either yes or no, and it is the job of the algorithm to come up with the correct answer. The curious thing about the TSP decision problem is that, while it might be very hard to find a solution, it is easy to discover if the solution is correct: just add up the distance between the cities in the given order. If the distance is less than 350 miles, then you have a solution. Such a solution is called a *certificate*.

(We've seen decision problems before: the Halting Problem is also a decision problem. Specifically, it is a decision problem that is provably unsolvable.)

A more complex version of TSP is known as an *optimization problem*: find the *best* possible solution. If you have an efficient way to solve a decision problem, you can efficiently solve the optimization problem by increasing the time that it takes by another factor of $log(n)$ by using binary search. Here, we could start by solving the decision problem for 300 miles. If the answer is yes, we try to solve the decision problem for 150 miles, if the answer is no, we try to solve the decision problem for 600 miles, and so on. Eventually we will find the optimal decision. (There are much more efficient ways

[35] If you could consider a billion (10^9) combinations every second, it would take 2.43 billion (2.43×10^9) seconds to find the answer. That's 77 years.

[36] Aaronson, *Quantum Computing since Democritus* (2013).

to solve the TSP optimization problem, but they are beyond what is needed here.)

In 1959 computer scientists Michael Rabin and Dana Scott proposed a model for a theoretical computer that made it easy to write algorithms for solving problems like TSP. They called it a *nondeterministic machine*;[37] today we call these creations-of-the-mind *nondeterministic Turing machines* (NTM).[38] The idea is that such a machine can explore all possible solutions simultaneously: when the right solution is found, the NTM recognizes that solution as the correct one.

Another way to conceptualize the NTM's theoretical module is to imagine that an NTM is just an ordinary computer that is equipped with a special module called `CORRECT_GUESS` that always guesses correctly.

In their paper, Rabin and Scott show that NTMs are no more powerful than conventional, deterministic Turing machines, but for many problems, the description of how to solve it is shorter when the write-up uses a NTM than the equivalent TM. That is, the two models are *mathematically* identical in the kinds of problems that they can and cannot solve.

To understand why TMs and NTMs are mathematically equivalent, but why it is easier to write up the program for a NTM, consider a program that factors a number N into two factors P and Q. The program on an NTM is simple:

```
ALGORITHM NTM_FACTOR(N):
  ℤ⁺ ← SET OF POSITIVE INTEGERS
  FOR ALL POSSIBLE P ∈ ℤ⁺, Q ∈ ℤ⁺:
     (P,Q) ← CORRECT_GUESS(P,Q, GIVEN (P × Q = N))
     RETURN (P, Q)
```

That is, the program tells the computer to correctly guess P and Q given that $P \times Q = N$ and that P and Q are integers.

If this looks like cheating, well … it is! Nondeterminism is all about cheating. The breakthrough insight of the 1959 paper is that one is allowed to cheat and not design algorithms if one does not care how long those algorithms take to complete.

[37] Rabin and D. Scott, "Finite Automata and Their Decision Problems" (1959).

[38] Rabin and Scott's article variously refers to the machine that they created as *nondeterministic machines* and *nondeterministic automata*, but for our purposes, we can take the article as describing NTMs as well.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

There are a lot of ways to find two factors of a number N. Here is one that is both naïve *and* inefficient:

```
PROGRAM NAIVE_FACTOR(N):
10 P ← 2
20 Q ← INTEGER( N ÷ P)
30 IF P × Q = N:
   RETURN (P, Q)
40 P ← P + 1
50 IF P > N ÷ 2:
   ABORT
60 GOTO 20
```

This program uses an approach called *trial division*. It tries to divide N by every number from 2 up to $\frac{N}{2}$. If it finds a number which evenly divides N, it returns that number and N divided by that number. If it doesn't find that number, it aborts.

Another way to describe this program is to say that it takes a *brute force* approach to the problem of factoring: it just tries every possible solution and stops when it finds one that works. This is the reason why TMs and NTMs are mathematically equivalent.

A common misconception about quantum computers is that they cheat in this way. They do not: *quantum computers are not NTMs*. Indeed, for a long time Scott Aaronson's blog had the tagline, "If you take just one piece of information from this blog: Quantum computers would not solve hard search problems instantaneously simply by trying all the possible solutions at once." Quantum computers can perform some functions dramatically faster than classical computers because of the algorithms discovered for certain problems. In some cases, these algorithms are just somewhat faster than classical counterparts. And yet in others, quantum computers will offer no real advantage over fast classical computers.

3.5.4 NP-Complete and NP-Hard

In 1971 Stephen Cook, a professor at the University of Toronto, presented a paper at the Third Annual ACM Symposium on the Theory of Computing that contained a startling discovery: any problem that could be solved by an NTM in polynominal time can be reduced to a specific *NP* problem called SATISFIABILITY.

SATISFIABILITY asks if there is an arrangement of Boolean variables that can solve a particular equation. Boolean variables can

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

have the value of TRUE or FALSE; a Boolean equation combines these variables with the operators AND, OR and NOT. So if A and B are Boolean variables, a simple instance of the SATISFIABILITY problem is:

```
SATISFIABILITY PROBLEM 1:
CHALLENGE:  (A AND B) IS TRUE
```

In this case, it is satisfied if A is TRUE and B is true. Here is the certificate:

```
SATISFIABILITY PROBLEM 1:
  CHALLENGE: (A AND B) IS TRUE
  SOLUTION:
    A: TRUE
    B: TRUE
```

Here is a problem that cannot be satisfied:

```
SATISFIABILITY PROBLEM 2:
   (A AND B) AND (NOT B) IS TRUE
```

This problem can't be satisfied, because the first clause can only be TRUE if both A and B are TRUE, while the second clause can only be TRUE if B is FALSE.

Cook's paper was astonishing, because it showed that any problem that can be solved in polynomial time on a nondeterministic Turing machine can be transformed into a SATISFIABILITY problem that can be solved in polynomial time. The following year, Richard Karp published a paper showing that 21 other problems have this property, including the TSP decision problem. This means that any given TSP decision problem can be quickly rewritten as a Boolean SATISFIABILITY problem. Conversely, any SATISFIABILITY problem can be rewritten as a TSP decision problem. If you can come up with a general solution for efficiently solving a SATISFIABILITY problem, you can solve TSP. If you can efficiently solve any TSP, you can efficiently solve SATISFIABILITY. Today this property is called *NP-complete*.

Since 1971, computer scientists have proven that hundreds of similar problems, including the traveling salesperson problem, are also NP-complete. On the positive side, this means that a solution to one of these problems could be easily repurposed to solve the others: a good solution to TSP can be used to solve packing problems, for example. But no such solution has ever been found, and many

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

researchers suspect that no such solution exists. Indeed, after the discovery of NP-completeness in 1971, many theoreticians thought that within five or ten years there would be a proof showing that problems in *P* (like sorting) are fundamentally easier than problems in *NP* (like TSP). But nobody could create such a proof.

Today, after 50 years of searching, computer scientists still lack proof that *P* and *NP* are fundamentally different kinds of problems. This is astonishing, because we have problems that are clearly easy, such as sorting a list of numbers into ascending order, and problems that are clearly hard, like solving complex Sudoku puzzles. Sorting is clearly in P, because there are algorithms of polynomial complexity that sort. Sudoku, meanwhile, is NP-complete. That is, there is no efficient algorithm for solving Sudoku, but there is an efficient algorithm for turning any other NP-complete problem *into* a Sudoku problem and vice versa. Perhaps there is some trick to solving Sudoku problems, just waiting there for someone to find it. Alternatively, there may be a proof that Sudoku is actually quite hard. And yet … nothing, even after 50 years of trying.

Even more infuriating, there are a few problems that were thought to be hard, yet turned out to be easy. One such problem is *primality testing*. Primality testing means to take a number and determine if it is a prime number or a composite. For decades the computing world had a fast, probabilistic primality test that could determine with high probability if a number was prime or not, but there was no fast *deterministic* test that could determine in a reasonable amount of time if a number was prime or not.[39] Then in 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena at the Indian Institute of Technology Kanpur announced the discovery that "PRIMES is in P,"[40] which presented a polynomial time algorithm for primality test-

[39] Note that primality testing is fundamentally a different problem than factoring. With primality testing algorithms it is relatively straightforward to take a thousand-digit number and determine in seconds if the number is prime or not. However, these primality testing algorithms do *not* yield the factors of the number that is being tested. This is similar to the fact that you can tell quickly if a number is divisible by 3 – just add up all the digits, and then take the resulting number and add up all the digits, and keep going until you have a single digit. If that digit is 3, 6, or 9, the original number was divisible by 3.

[40] Although the preprint of the article was published on the Internet in 2002, the formal article wasn't published for two more years, finally appearing as (Agrawal, Kayal, and Saxena, "Primes Is in P" (2004)).

ing. This means that you can now take any number and determine quickly if the number is prime or not.

Primality testing is one thing, but what about factoring? Is that hard? Or is there some hidden algorithm waiting to be discovered? We don't know. Although factoring is clearly in the complexity class *NP* – there is a simple *NP* algorithm for factoring any number *N* – efforts to prove that it is or is not NP-complete have failed.[41] Perhaps a polynomial-time algorithm for factoring exists just out of reach, about to be discovered.[42] Today most computer scientists believe both that $P \neq NP$ and that factoring is not NP-complete, but this is a matter of faith, not of proof. For more information, see Section 3.5.6 (p. 116).

In addition to NP-complete problems, there is another complexity class called NP-hard. NP-hard problems are problems at least as hard as NP-complete problems, but possibly harder. One way to think of these problems is to consider the set of problems for which it is not obvious how to create a certificate. These problems might be fundamentally harder than NP-complete problems, or perhaps there is a way to efficiently create a certificate, and it just hasn't been discovered (yet).

Consider the game of chess. Assuming that it is white's turn to move, any given board position may be a winning position for white, meaning that there is a specific sequence of moves and counter-moves that white can play for which every possible response by black always leads to a victory for white or a draw. Likewise, any given board position may be a losing board for white, meaning that no matter what white does, black can always either win or achieve a draw. It is not clear what a certificate for Chess would look like. The most straightforward certificate would be a list of every possible move by white, followed by every possible response by black, and so on. But such a certificate would grow exponentially large with respect to the

[41] The simple *NP* algorithm for factoring any number N is to try all possible combinations of the numbers a and b such that $1 < a \leq b < N$ until you find a value of a and b such that $a \times b = N$. If you can find such a pair of numbers, then those are the factors.

[42] It turns out that factoring can be done in polynomial time on a quantum computer. Such algorithms are said to be in the complexity class *BQP* (bounded-error quantum polynominal time). Such algorithms are discussed in the next chapter. Perhaps one day someone who learns enough about quantum computing will come up with a fast factoring algorithm that runs quickly on conventional computers. If such an algorithm is found, that will prove that factoring is in *P*.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

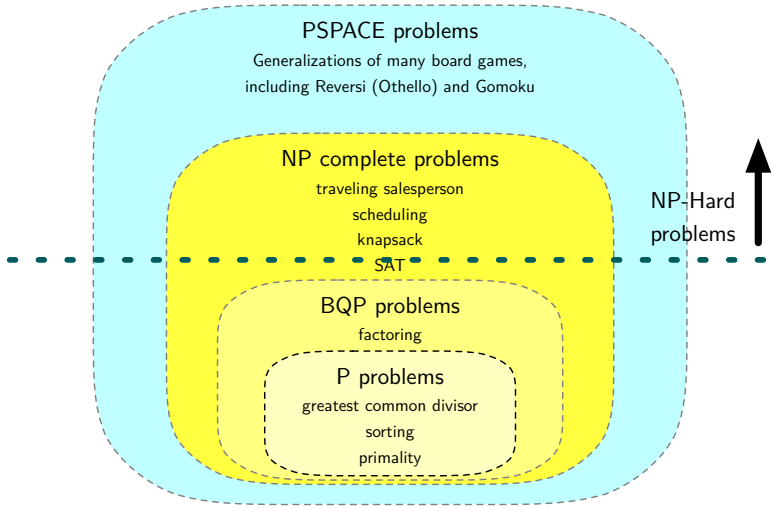


Figure 3.6. The *P*, *BQP*, *NP* and *PSPACE* complexity spaces as they are thought to be if $P \neq BQP \neq NP \neq PSPACE$. It is currently unproven if *PSPACE* and NP-complete problems are in the same complexity class, or if there is a partition between the two. Likewise, it is unproven if *NP* and *BQP* are in the same complexity class, and if *BQP* and *P* or in the same class. If $NP = P$, then *NP*, *BQP* and *P* are all in the same class. However, it is straightforward to prove that *P* and *PSPACE* are in different complexity classes.

number of pieces on the board, and it would therefore take exponentially long to check. In fact, the only way to check such a certificate would be to regenerate the certificate and prove it for yourself, and so this list-of-all-possible-moves-certificate doesn't actually accomplish its objective of being a certificate – that is, it doesn't save any time when you go to check it. Chess is said to be in the complexity class PSPACE, meaning that it requires polynomial space to solve – in this case, that space holds all of the possible chess games. In fact, Chess is said to be PSPACE-complete, actually meaning that all PSPACE problems can be reduced to the problem of finding a winning chess game (assuming you have an infinite number of chess pieces and an arbitrarily large chess board in which to express your problem). Perhaps there is a dramatically more efficient representation for all possible moves in a specific Chess instance; perhaps you will discover it.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

3.5.5 NP-Complete Problems Are Solvable!

Just because a problem category is NP-complete doesn't mean that a specific instance of a problem in that category is impossible, or even hard, to solve. SATISFIABILITY is NP-complete, but problems 1 and 2 above are both trivial to solve. Indeed, TSP has been recognized as an important problem for more than 100 years,[43] and there are a growing number of approaches for solving the problem faster, such that in 2004 a challenge problem with 85 900 cities was solved in 136 years of computer time on a cluster of 2.4 GHz computers. (Because the program can be parallelized, it could be on a single computer for 136 years, or on 136 computers in 1 year, or on 1360 computers in 37 days.[44]) The actual computation was performed on a mix of computers between February 2005 and April 2006, because the TSP only ran when the computers were not being used for other purposes. In 2009 the group published a certificate proving that their solution was optimal: that certificate is 32.2 MB (uncompressed) and can be verified in just 569 hours.[45]

As mentioned above, the field of operations research really got going during World War II. One of the exciting early developments was the discovery of the simplex algorithm, an approach for optimizing a system of linear equations. Although simple problems can be solved exactly using symbolic mathematics, many optimization problems are solved in practice using iterative numerical methods – that is, the computer performs a series of computations, examines the results, and then repeats the computations many times in a row, with each iteration producing a more accurate result. Programs that can perform these kinds of optimizations are called, unsurprisingly, *optimizers*. Some optimizers are designed to solve a specific kind of problem, while others are general-purpose solvers, employing a broad range of algorithms and heuristics. The best optimizers today are commercial programs that cost thousands of dollars *per month* to run and save their users considerably more – according to one case study, Air France saves 1 percent of its fuel costs by using an optimizer to help assign planes to routes.[46]

[43] W. J. Cook, *In Pursuit of The Traveling Salesman* (2012).

[44] Applegate, Bixby, Chvátal, and W. J. Cook, *The Traveling Salesman Problem* (2006).

[45] Applegate, Bixby, Chvátal, W. Cook, et al., "Certification of an Optimal TSP Tour through 85,900 Cities" (2009).

[46] Gurobi Optimization, "Air France Tail Assignment Optimization" (2019).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

3.5.6 BQP, BPP, and Beyond

So here's where things stand in the Summer of 2021, as we get ready to send this book to the printer:

- The class *P* contains problems that can be solved in polynomial time with respect to the problem's size. That is, they don't get dramatically harder as the problem gets larger. For example, determining if a word is in a book is a problem that's in P: just look at every page in the book to verify that the word is not there. If a second book has twice as many pages, it will take you twice as long to check that book. Many common computer problems are in P, such as sorting a list of numbers or taking the square root of a number. It turns out that determining if a number is prime or not is also in P.

- The class *NP* are the problems that take exponentially longer to solve as the problem gets larger, but can be *verified* in polynomial time. Factoring is a good example: there is no fast way to factor a large integer like N, but if somebody gives you two small integers a and b and claims that $a \times b = N$, you can verify this pretty fast. Factoring is in *NP*.

- Some *NP* problems have a property called NP-complete. It turns out that SATISFIABILITY, the Traveling Salesperson Problem, Sudoku played on an arbitrarily large $n \times n$ grid, and many other problems are all fundamentally the same problem. By this we mean that a SATISFIABILITY problem can be transformed (in polynomial time) into a TSP, and vice versa. Transforming the problem is fast, but solving the transformed problem is still hard. On the other hand, this means that if we find a fast way to solve *any* NP-complete problem, we've identified a fast way to solve them all.

- It's unknown whether or not *P* and *NP* are actually the same class. There might be some clever way to transform an *NP*-complete problem into a *P* problem – that is, to solve it in polynomial time. If we find that way, then $P = NP$. Most computer scientists think that this is highly unlikely, but even after decades of trying, nobody has been able to prove that $P \neq NP$. We write this confusion as: $P \stackrel{?}{=} NP$.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

BQP is a complexity class that is conjectured to be between *P* and *NP*. As we will see in Chapter 5, there are a growing number of problems that take exponential time to solve on a classical computer, but which can be solved quickly, in polynomial time, on a quantum computer. This is the class *BQP*, short for bounded-error quantum polynomial time. The proofs that these algorithms are correct typically involve a combination of quantum mechanics and number theory, but they are irrefutable – that is, they are irrefutable if you believe in mathematics and quantum mechanics. We used the word *conjectured* at the start of this paragraph because if $P = NP$, then $P = BQP = NP$. But it might also be the case that $P = BQP \subseteq NP$ or that $P \subseteq BQP = NP$. We just don't know!

A second complexity class that is conjectured to be between *P* and *NP* is *BPP*, the bounded-error probabilistic polynomial time complexity class. *BPP* is like *BQP*, but instead of using a quantum computer, the algorithms are run on a conventional computer (a Turing machine) that has access to a true random number generator. It turns out that there are many algorithms that can run much faster if they have access to truly random numbers: these are called *randomized algorithms*. Until 2002 primality testing was known to be in *BPP*, because there was a randomized algorithm that did an arbitrarily good job determining if a number is prime or not. Then in 2002, Agrawal et al. developed an algorithm that can test if a number is prime or not in polynominal time.[47] This was a huge breakthrough. However, the algorithm is slower than the randomized algorithm, so in practice the randomized algorithm is typically used in preference to the 2002 algorithm.

Quantum mechanics gives quantum computers an unlimited supply of perfectly random numbers so *BQP* necessarily contains *BPP*: that is, every problem in *BPP* can be solved in polynomial time by a quantum computer. But we don't know if *BPP* is the same as *BQP* or contained in BQP. We write this mathematically as:

$$BPP \stackrel{?}{\subseteq} BQP \tag{1}$$

This means that we can write the complexity theory that we covered above succinctly as:

$$P \stackrel{?}{\subseteq} BPP \stackrel{?}{\subseteq} BQP \stackrel{?}{\subseteq} NP \tag{2}$$

[47] Agrawal, Kayal, and Saxena, "Primes Is in P" (2004).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

And we have just gotten started! Today there are hundreds of complexity classes that have been formally defined – the online "Complexity Zoo" (www.complexityzoo.net) listed 545 such classes as of April 2021. (The website also has an easier-to-digest "petting zoo" that has just 17 complexity classes.) The good news is that not all of these classes little question marks over their relations: recall that it's straightforward to prove that $P \subset PSPACE$. But we won't prove it here! To see that proof, and many others, we recommend *Introduction to the Theory of Computation, 3rd Edition*.[48]

3.6 Computing Today

More than any other human technology, electronic computation has undergone phenomenal changes since its inception roughly 80 years ago. That improvement has come both from roughly a trillion-fold increase in the speed of computation and storage, as well as a speedup in the efficiency of algorithms that is surprisingly difficult to measure. But starting in the early 2000s, technology trends changed abruptly:

- Many of the tricks that semiconductor companies had used to speed up their computers since the 1960s started to sputter out. Companies like Intel responded by putting two, four, eight or more general-purpose computers on a single chip, what is now called *multi-core systems*. Companies like NVidia responded by putting hundreds and then thousands of restrictive, special-purpose cores on graphics cards, called *graphical processing units (GPUs)*. Programmers responded by adapting software to use this more difficult-to-program hardware.

- Companies like Amazon, Google, and Yahoo developed and deployed workable approaches for orchestrating thousands of individual computer systems to solve a individual complex problems. These approaches, alternatively called *cluster-computing*, *grid-computing*, and *warehouse-scale computing*, first appeared in the 1990s in the world of scientific computing, where engineers created systems with dozens and then hundreds of racks, each filled with very expensive, very reliable machines. The big breakthrough in the 2000s was the realization that companies could achieve better price-performance ratios by using commodity hardware. In today's warehouse-scale computing, each

[48] Sipser, *Introduction to The Theory of Computatio* (2012).

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

individual system isn't as fast or as reliable as the high-end systems used in scientific computing, but the individual computers are so much cheaper that many more computers can be purchased for the same cost, and fault-tolerant software can automatically reschedule work on a different computer if there is a hardware failure.

- Corporations that previously bought and ran their own computer systems transitioned to renting slices of computers at shared data centers. This approach, called *cloud computing*, gave organizations access to far more computing than was previously possible. The reason for this is that most organizations (and individuals) do not need a steady amount of computing power: they need it in bursts. Thus, just as it is more economically efficient for a home-owner wanting to dig a trench to rent rather than purchase an excavator (and perhaps an operator), in like manner, it is more efficient for a business that needs to solve a big problem to rent a few thousand virtual machines for a week, than to purchase a few dozen machines and run them for six months or a year.

The rate of technology change accelerates because one of the things that engineers can do with faster computers is create faster computers. For example, computer programs running on today's top-of-the-line integrated circuits not only help engineers design the next generation systems – today's computers can also *simulate* next years' systems to find out if the systems will work when they are finally constructed. Even though such simulations run significantly slower than will the future chips, they still help engineers find problems with the chips while they are still being designed, which saves money, shortens design cycles, and allows engineers to pursue more aggressive designs.

This feedback loop, what some people call a *virtuous circle*, is the reason that computers have become a trillion times faster, while aircraft and cars travel no faster today than in the 1960s: faster, more powerful vehicles don't make it possible to build faster, more powerful vehicles.

3.7 Conclusion

For most of its early history, computing has been a tool of governments to solve the kinds of problems governments have. Govern-

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press

ment and academic research in computing led to its adoption in other data-intensive activities. The trends of democratization of computing services through parallelization, cloud, and eventually the personal computer, brought these devices into our daily lives in unforeseen, wonderful ways.

The path and future of quantum computing could share characteristics with those of classical computing, but with important differences. Like classical computers, quantum computers need patronage from well-resourced and determined actors, and this often requires that government/military problems are on the front burner for applications. Classical computers experienced successive generations of speedups in hardware improvements from the relay, to the vacuum tube, to the transistor. Since the 1960s, classical computing has been transistor based. Quantum computing is still in the relay-vacuum tube stage and needs a breakthrough on the level of the transistor to scale up.

The introduction to complexity theory in this chapter lays the foundation for elucidating the kinds of applications that quantum computing will pursue most effectively. The press often focuses upon cryptanalysis as the problem that quantum computers will solve. However, complexity theory shows that much more interesting, yet more difficult-to-understand challenges, with far-reaching social implications, will be important domains for quantum computing.

https://doi.org/10.1017/9781108883719.007 Published online by Cambridge University Press