C

Computational Complexity

An algorithm is a finite answer to an infinite number of questions

— Stephen Kleene

Computational complexity theory is the branch of theoretical computer science that is concerned with quantifying the resources needed to solve problems with algorithms. It asks questions such as "How much time is needed to multiply two integer numbers of n bits each?", "Do you need more memory space to solve a problem than to check its solution?", or "Is randomness useful in computational tasks?".

In this brief introduction to computational complexity, we will focus mainly on the concepts involved in estimating how much time is required to solve certain problems. For a thorough treatment of this and other topics (including space or memory complexity, the role of randomness in computation, approximation algorithms, and other advanced matters), you can check standard computational complexity books such as the ones by Sipser [26], Papadimitriou [102], or Arora and Barak [103].

To study the kind of questions posed in computational complexity theory, we need first to introduce a computational model that allows us to measure computation time, memory, and other resources. The usual choice is that of **Turing machines**. It is beyond the scope of this book to mathematically define what Turing machines are (for the details, check the books cited in the previous paragraph), but let us at least give an informal description so you can understand how we can use them to model computational tasks and to measure the resources involved in solving problems with them. Please notice that different textbooks use slightly different definitions of Turing machines, but it is straightforward to show that they are all equivalent in power.

A few words on Turing machines

A Turing machine is a (theoretical) device that has a (potentially infinite) **tape** divided into **cells**. Each of these cells can store a symbol from a finite and fixed number of possibilities (usually, 0, 1, and a “blank” symbol to denote an empty cell). The machine also has a **head** that, at any given moment, is scanning one of the tape cells. Additionally, the machine is in a **state** (also from a finite number of fixed options) at any step in the computation.

The machine has a list of instructions that, depending on the machine’s state and the content of the cell that the head is scanning, tell the machine what it should do next. This can involve changing the machine state, writing a different symbol on the cell that is being scanned, and moving the head one cell to the left or to the right. For instance, one such instruction could be “If the state is q_2 and the symbol being read is 1, change the state to q_5, change the symbol to 0, and stay in the same cell,” while another could be “If the state is q_0 and the symbol is 0, change the state to q_1, leave the symbol unchanged, and move the head one cell to the right.”

Important note

A Turing machine is a (theoretical) device that has an unbounded tape divided into cells and a head that scans one of those cells. At any given moment, the machine is in an internal state from a finite number of possibilities. The instructions of the machine specify, depending on the machine state and the content of the cell that the head is scanning, what the next state is, the new content of the cell, and the action of the machine (move left, move right, or stay, for instance).

In order to perform a computation, the input is given as a finite string of symbols on the tape (the rest are left blank). Then, the Turing machine operates in the following way: it starts in a predefined initial state and with its head scanning the first symbol of the input; then, it changes its state, tape content, and head position following its instructions in discrete steps. Eventually, the machine can stop because it reaches a predefined, halting state. If the machine stops, the output of the computation is the string of symbols written on the tape.

To learn more…

It is not guaranteed that a Turing machine will stop for all its inputs. In fact, it can be proved that determining whether a Turing machine will eventually stop with a given input (what is usually called the **halting problem**) is unsolvable in a very precise way: there is no algorithm that can give the correct answer for every possible Turing machine and every possible input. Check the book by Sipser [26] for a proof of this amazing fact.

Turing machines may seem like too simple a model, but it can be proved that any computation that can be carried out with any other reasonable computational model can also be carried out with a Turing machine (maybe with some slowdown). For instance, it is rather straightforward to prove that if we extend Turing machines by giving them multiple tapes (**multi-tape Turing machines**) or the possibility of non-deterministically choosing among several instructions for the same state-symbol situation (**non-deterministic Turing machines**), the new devices aren't more powerful than our original single-tape, deterministic Turing machines (again, see the book by Sipser [26] for all the details). The same happens if we consider models that are much closer to the actual architecture of modern computers, such as the **Random-Access Machines** model (see Section 3.4 in the book by Savage [104]), or even models, such as that of **while-Programs** (see the book by Kfoury, Moll, and Arbib [105]) that are based on common programming languages.

This has led to the firm belief that Turing machines indeed formally capture the informal notion of what an algorithm is. This fact is usually known as the **Church-Turing thesis**.

Measuring computational time

We can say that the Church-Turing thesis is simply stating that, if you are only interested in identifying which tasks can be solved algorithmically and which cannot, you can just use any of a wide number of equivalent models: single-tape Turing machines, multi-tape Turing machines, non-deterministic Turing machines, Random-Access Machines, while-Programs, and many, many others. Each of them will give you exactly the same power.

But be cautious! If you care about the resources needed to carry out the computations (and that is what computational complexity is all about), then the choice of the model can be important. So let's fix, for now, the single-tape Turing machines (the ones that we have described informally in the previous section) as our computational model. In this way, we can easily measure the time needed to carry out a certain computation with one of these Turing machines as the number of steps that it must take to complete it.

That works well for a fixed Turing machine with a particular input, but we are usually more interested in analyzing how the running time grows with the size of the input than we are in finding concrete running-time values for concrete problem instances. For example, we could be interested in knowing whether the time needed for a certain task grows so rapidly that it quickly becomes unfeasible to solve the problem when the input size becomes moderately big.

For this reason, we will define the running time of a Turing machine as a function of the input length, not as a function of the particular input. Namely, the running time of a Turing machine M is a function T that takes as input a non-negative integer n and returns the maximum number of steps that M performs with an input x of n bits before it stops. Notice that this is a worst-case definition of running-time: it is defined in terms of the string that needs the most time in order to be processed. Note also that, if a machine does not stop for some inputs, its running time for inputs of those lengths will be infinite. This is not a problem for our purposes, because we will only consider machines that always stop.

Important note

The running time of a Turing machine M is a function T such that $T(n)$ is the maximum number of steps that M performs when given an input of length n.

For other computational models, running times can be defined in analogous ways. For instance, for multi-tape Turing machines, the running time is again measured as the maximum number of steps performed on inputs of size n. For computational models that use idealized programming languages (the while-Programs model, for instance) or abstract architectures (the Random-Access Machines model), running time can be defined as the maximum number of basic instructions (setting a variable to zero, incrementing a variable, comparing the value of two variables…) executed with inputs of size n.

Asymptotic complexity

In order to compare different running times associated with different Turing machines, it is convenient to perform some simplifications. We usually do not care about whether the running time of a Turing machine is exactly $T_1(n) = 4321n^2 + 784n + 142$ or, rather, $T_2(n) = n^3 + 3n^2 + 5n + 3$. In fact, we are more interested in whether $T(n)$ grows roughly like n^3 or like n^2, because this implies a qualitative difference: for values of n that are big enough, any polynomial of degree 3 grows more rapidly than any polynomial of 2. In the context of computational complexity theory, we would always prefer a $T(n)$ that grows as n^2 over one that grows as n^3, because its behavior for big inputs (its asymptotic growth, in other words) is better.

This intuitive idea is captured by the famous **Big O notation**. Given two time functions $T_1(n)$ and $T_2(n)$, we say that $T_1(n)$ is $O(T_2(n))$ (and we read it is as "$T_1(n)$ is Big O of $T_2(n)$") if there exist an integer constant n_0 and a real constant $C > 0$ such that for all $n \geq n_0$ it holds that

$$T_1(n) \leq CT_2(n).$$

For instance, you can check that $4321n^2 + 784n + 142$ is $O(n^3 + 3n^2 + 5n + 3)$.

The main idea behind this definition is that if $T_1(n)$ is $O(T_2(n))$, then the growth of T_1 is not worse than that of $T_2(n)$. For example, it is easy to prove that n^a is $O(n^b)$ whenever $a \leq b$ and that n^a is $O(2^n)$ for any a. But, on the other hand, n^b is not $O(n^a)$ and 2^n is not $O(n^a)$. See *Figure C.1* for an example with linear, quadratic, cubic, and exponential functions. Notice how the exponential function eventually dominates all the others despite having 10^{-4} as its coefficient.

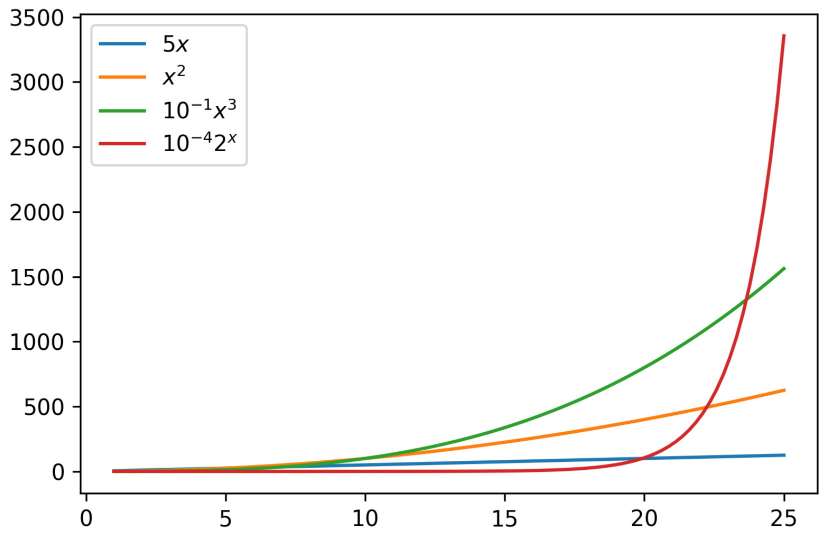

Figure C.1: Growth of linear, quadratic, cubic, and exponential functions

Important note

Given two non-negative functions $T_1(n)$ and $T_2(n)$, we say that $T_1(n)$ is $O(T_2(n))$ if there exist n_0 and $C > 0$ such that

$$T_1(n) \leq CT_2(n)$$

for every $n \geq n_0$.

Big O notation is extremely useful to estimate the behavior of running times without having to focus on small, cumbersome details. If the running time of a Turing machine is $4321n^2 + 784n + 142$, we can just say that it is $O(n^2)$ and forget about the particular coefficients in the time function. This is also the reason why we can abstractly think about the number of steps and not, for example, milliseconds. The particular amount of time that each step takes is a constant that will be "absorbed" by the Big O notation.

However, this comes at a price. A running time such as $10^{100}n^2$ is certainly $O(n^2)$. But it is not preferable to n^3 unless $n > 10^{100}$, something that will never happen in practical situations, because 10^{100} is much, much bigger than the number of atoms in the visible universe. So use this notation wisely: with Big O comes Big Responsibility.

P and NP

As we mentioned at the beginning of this appendix, computational complexity theory studies the amount of resources needed to solve problems with algorithms. So far, we have focused on how to mathematically define the notion of algorithm with the help of Turing machines and on how to measure the time needed to perform computations with them. Now, we turn our attention to defining computational problems and classifying them according to the time they take to be solved. That is, we will think in terms of their inherent complexity and not in terms of specific algorithms.

In computational complexity theory, a **problem** consists of an infinite number of instances or inputs for which an output value needs to be returned. For example, we may be given two natural numbers and asked to compute their product. Or we may be given a graph and asked to check if it has a Hamiltonian path or not. In both cases, the number of possible inputs is infinite and there is a well-defined output or answer associated with each such input.

Problem instances are usually encoded as binary strings in some way. For example, we can represent a natural number by its binary expansion or a graph by (the concatenation of the rows of) its adjacency matrix. In the same way, outputs can also be represented by binary strings. Consequently, a problem can be identified with a function that takes a binary string as its input and returns a binary string as its output. But a Turing machine does exactly that: it receives binary strings as inputs and returns binary strings as outputs. This allows us to study which problems can be solved with Turing machines and how much time is needed to solve them.

In computational complexity, the simplest category of problem that we can consider is that of **decision problems**, in which the output is a single bit (we usually identify 1 with "true" and 0 with "false"). Examples of decision problems include determining whether a natural number m is prime, determining whether a graph has a Hamiltonian path, and determining whether a Turing machine stops for all its inputs.

We say that a Turing machine is a **decider** for a decision problem if, given as input a binary string representing an instance of the problem, it eventually stops and returns the correct output (0 or 1) for that instance. In that case, we also say that the Turing machine **solves** or **decides** the problem. There exist deciders for the problems of determining whether a number is prime and of determining whether a graph has a Hamiltonian path, but not for the problem of determining whether a Turing machine stops for all of its inputs (this is a consequence of the unsolvability of the halting problem that we mentioned earlier).

Once we know that a problem has a decider, we can try to further refine its classification by taking into account the resources used by the decider. This leads, for instance, to the definition of the famous P (short for "polynomial time") class. We say that a decision problem A is in P if there exists a decider for A that runs in polynomial time. That is, there exists a Turing machine D that decides A and whose running time $T(n)$ is $O(n^a)$ for some non-negative integer a. Notice that, for a problem to be in P, it is enough to find one polynomial-time decider for it. However, in order to show that a decision problem A is not in P, we need to prove that no Turing machine running in polynomial time is able to decide A. This is usually much, much harder to do.

As an example, a celebrated result by Agrawal, Kayal, and Saxen [106] shows that the problem of determining whether a natural number is a prime is indeed in P. Other, simpler examples of problems in P include checking whether a number is a perfect square or checking whether a binary string is a palindrome (that is, it reads the same from left to right and from right to left). However, for the problem of determining whether a graph has a Hamiltonian path, we do not know whether it is in P or not. We very strongly believe that it is not in P, but despite the best efforts of thousands of mathematicians over several decades, we still can't prove it.

Important note

We define P as the class of decision problems that can be solved with Turing machines in polynomial time.

Actually, P is interesting for several reasons. First, it is quite robust. We have defined it in terms of the computation time required by deciders that are single-tape Turing machines. However, if we had chosen another computational model such as, for instance, multi-tape Turing machines, then we would have arrived at exactly the same set of problems. This is so because it is possible to simulate a multi-tape Turing machine with a single-tape Turing machine with just a polynomial overhead in running time. The same is true for any other reasonable (classical) computational model, so although the particular running time might differ from one model to another (say $O(n^4)$ with single-tape Turing machines and $O(n^2)$ with 2-tape Turing machines), one will be polynomial if and only if the other is.

What is more, P seems to capture quite well the notion of a problem being efficiently solvable. It is true that in P we allow running times such as n^{1000}, which can hardly be deemed as efficient. However, the running time of naturally-occurring problems that we can prove to be in P is typically much more tame, such as $O(n^2)$ or $O(n^3)$. Moreover, if a decision problem is not in P, then the running time of any of its deciders will grow faster than any polynomial (at least, for an infinite number of its inputs). And that is something that we can unequivocally classify as not efficient at all.

Another central class of problems in computational complexity is NP. It is, again, a class of decision problems. But, in this case, the defining property is not that we can solve them efficiently (as in the case of P) but that we can check their solutions with an efficient algorithm. To make this idea formal, we say that a problem A has a **polynomial-time verifier** if there exists a Turing machine V that runs in polynomial time and a polynomial q with the two following properties:

- If x is an instance of problem A of size n for which the answer is "true," then there exists a binary string y of length at most $q(n)$ such that V on input (x, y) returns 1. The string y is usually called a **witness**, a **certificate**, or a **proof** for x.
- If x is an instance of problem A of size n for which the answer is "false," then for every binary string y of length at most $q(n)$, V on input (x, y) returns 0.

This definition is a little bit convoluted, so let's analyze it in detail. The idea here is that for an instance x of A whose answer is positive, we can find a certificate y that is not long (its length is polynomial in the size x) and that we can check when we are given y together with x, with an efficient algorithm. However, for instances whose answer is negative, there is no such certificate. Note also that the total running time of V on (x, y) is polynomial in the length of x, because V runs in polynomial time in its whole input and y has a length that is polynomial in x. Hence, this definition really captures the notion of checking that the answer to x is positive (through certificate y) with an efficient algorithm.

With this notion at our disposal, we can now define NP as the class of decision problems for which there exists a polynomial-time verifier.

To learn more…

An alternative, but equivalent, definition of NP can be given in terms of non-deterministic Turing machines. In fact, NP is short for "non-deterministic polynomial time." You can find all the details in Sipser's book [26].

Let's discuss an example to illustrate this definition. The problem of determining whether a graph has a Hamiltonian path is in NP. The certificate y can, in this case, be just a Hamiltonian path in the graph. Indeed, it is easy to write a program (in Python, for example) that, given a graph represented by x and a sequence of vertices represented by y, checks whether y is a path in x that visits all the vertices in the graph. Moreover, we can easily do this computation in polynomial time and the certificate is always of size linear in the number of graph vertices. As required, for graphs that have a Hamiltonian path, there exists at least a certificate. However, for graphs without Hamiltonian paths, no y will make the verifier output 1. If needed, we could translate our algorithm into Turing machine instructions; it is a tedious process, but it has no real difficulty.

Important note

NP is the class of decision problems whose solution can be verified with Turing machines in polynomial time.

Similar arguments can be given to prove that many important problems are in NP, including determining whether a Boolean formula is satisfiable, determining whether a graph is 3-colorable, or determining whether a graph has a cut of size bigger than a given integer k. The certificates for them can, of course, be a satisfying assignment, a 3-coloring of the graph, and a cut of size bigger than k. All of them are of a size comparable to the problem instances they certify and can be checked efficiently with obvious procedures.

Additionally, any problem in P is also in NP. This is easily proved. By definition, a problem A in P has a decider. But we can directly use this decider to obtain a verifier for A: we only need to ignore the candidate certificate y and compute the answer with the decider itself.

If the machine knows how to solve the problem in polynomial time on its own, it does not need any external help!

So, we know that P is contained in NP. And it seems like we should be able to prove that they are different, because there must be problems whose solutions we can check efficiently, but for which it is impossible to find those same solutions in a reasonable amount of time, right? Well, it turns out that this is by no means an easy task. In fact, it is literally the million-dollar question!

Determining whether $P = NP$ is one of the seven Millennium Problems selected by the Clay Mathematics Institute in 2000 as the most important open questions in all of mathematics (for an accessible account of the Millennium Problems, check the book by Keith Devlin [107]). Whoever is able to give proof showing that $P \neq NP$ or to show that every problem in NP is also in P, will receive a one-million-dollar prize and will become world-famous.

Important note

Every problem in P is also in NP. The question of whether there are problems in NP that cannot be solved in polynomial time is one of the most important open questions in all of mathematics.

Almost every expert in computational complexity believes that, in fact, $P \neq NP$. All the evidence points in that direction. And it certainly seems logical that *checking* a solution should be easier in general than *finding* a solution. However, no one has yet succeeded in proving that there are problems in NP that are not in P, and the most natural proof techniques have been shown to be insufficient (see *Section 6.5* in the epic book by Moore and Mertens [108]).

Hardness, completeness, and reductions

Although our current mathematical tools are not powerful enough to give satisfactory lower bounds on the resources needed by computational problems, we do know a good deal more about comparing the relative hardness of problems. The main concept used for that kind of comparison is what we call a **reduction**.

Intuitively, a reduction is a procedure to solve a problem from the solution to a different problem. We could say that we reduce solving problem A to solving problem B. So if we know how to solve B with an algorithm, we can use that algorithm and some additional computation to also solve A.

To put it more formally, consider two problems A and B, and imagine that we have an algorithm M_B that solves B. M_B is usually called an **oracle** for B. We say that A is **reducible** to B if we can solve A given an oracle for B. For instance, multiplying two numbers is reducible to adding two numbers: if we are given an oracle that adds numbers, we can use it to multiply by repeated addition.

Of course, when studying computational classes such as P and NP, we are interested in reductions that take a polynomial amount of time. But how can we capture that idea formally? Well, we can simply count each call to the oracle as just another step in the computation. Then, we say that a problem A is **polynomial-time reducible** to a problem B if, given an oracle M_B for B, we can solve any instance x of A with a total number of computational steps plus calls to M_B that is polynomial in the size of x. Another way of seeing this is imagining that we extend our Turing machines with the capability of computing M_B in a single step (these new devices are unsurprisingly called **oracle Turing machines**). Then, showing that A is polynomial-time reducible to B is the same as finding an oracle Turing machine (with an oracle for B) that solves A in polynomial time.

Notice that A being polynomial-time reducible to B has important consequences. The first one is that if B is in P, then A is also in P. This is so because, if B is in P, we can replace every call to M_B with an actual Turing machine that solves B and runs in polynomial time, making the total time involved in solving A also polynomial. This also implies that if A is not in P, then B cannot be in P either, because it would lead us to a contradiction.

Now, we say that a problem B is NP**-hard** if every problem A in NP is polynomial-time reducible to B. This means that B is at least as hard as any problem A in NP, because if we knew how to solve B efficiently, then we would also know how to solve A efficiently. And if at least one problem in A cannot be solved in polynomial time, that implies that B cannot be solved in polynomial time either.

Important note

A problem is NP-hard if every problem in NP is polynomial-time reducible to it.

Being NP-hard seems like a very strong property. Is it really possible for *every* problem A in NP to be reduced to a single problem B? As surprising as this may seem, we know of hundreds (if not thousands) of problems that occur naturally in practice and that are indeed NP-hard. A notable example is the problem of determining whether a Boolean formula is satisfiable or not, also called SAT. That SAT is NP-hard is the content of the famous Cook-Levin theorem (see the book by Sipser for a proof [26]). In *Chapter 3*, *Working with Quadratic Unconstrained Binary Optimization Problems*, we work with many NP-hard problems. For many other examples and much more on the concept of NP-hardness, you can check the classical book by Garey and Johnson [109].

In fact, it turns out that we can prove that SAT and other decision problems in NP have a property that is a bit stronger than NP-hardness known as NP-**completeness**. In order to discuss it, we first need to talk about a special type of reduction that is very useful when studying decision problems. We say that a decision problem A is **many-one reducible** to a decision problem B if there exists an algorithm F that transforms an instance x of A into an instance $F(x)$ of B with the property that the answer to x in A is positive if and only if the answer to x in B is positive.

Note that, in this case, we indeed have a reduction in the more general sense that we were discussing earlier. If we are given an oracle M_B for B, we can solve any instance x of A by computing $F(x)$ and applying M_B to $F(x)$. Here, we are using only one call to M_B, but in a general reduction, we can use M_B as many times as we see fit. Thus, a many-one reduction is a special case of a reduction. Additionally, in the case in which the transformation F can be computed in polynomial time, we say that we have a **polynomial-time many-one reduction**.

Important note

A polynomial-time many-one reduction of a decision problem A to a decision problem B is a polynomial-time algorithm F that takes instances x of A to instances $F(x)$ of B with the property that the answer to x in A is “true” if and only if the answer to $F(x)$ in B is “true.”

Now, we can actually define that subclass of NP-hard problems that we talked about before: the class of NP-**complete** problems. We say that a problem is NP-complete if it is both in NP and every problem in NP is polynomial-time many-one reducible to it. As we mentioned before, SAT, for example, is NP-complete. Other NP-complete problems include determining whether a graph is 3-colorable, determining whether the constraints of a binary linear program can be satisfied, determining whether a graph has a cut of size bigger than a given integer k, and many other natural decision problems.

NP-complete problems are central to the study of the $P \stackrel{?}{=} NP$ question because $P = NP$ if and only if at least one NP-complete problem is in P. So, you can focus on, say, just studying SAT. If you find a polynomial-time algorithm for it, then $P = NP$. If, on the contrary, you show that it is impossible to solve SAT in polynomial time, you have found a problem in NP that is not in P and then, immediately, you can conclude that $P \neq NP$.

Important note

A problem B is NP-complete if it is in NP and every other problem A in NP is polynomial-time many-one reducible to B.

There are, of course, NP-hard problems that are not NP-complete. This is the case, for instance, if you have an NP-hard problem that is not a decision problem (and, hence, cannot be in NP). Many problems that we study in *Chapter 3*, *Working with Quadratic Unconstrained Binary Optimization Problems*, fall under that category. For instance, finding a minimal coloring for a graph is clearly NP-hard. If you knew how to solve this problem efficiently, then you could also determine whether a graph is 3-colorable (you just need to compute the minimal coloring and check whether its number of colors is at most 3). But checking whether a graph is 3-colorable is NP-hard and, thus, finding a minimal coloring is also NP-hard.

Many other examples of problems that are optimization versions of NP-complete problems are also NP-hard, including determining the maximum number of clauses that can be simultaneously satisfied in a Boolean formula in conjunctive normal form (the MAX-SAT problem), finding a maximum cut in a graph (the Max-Cut problem), finding a minimum-cost solution of a binary linear program, or solving the Traveling Salesperson problem. However, none of them is NP-complete because they are not in NP: they are not decision problems to start with and, moreover, it is far from clear that you could check efficiently that a candidate solution is, indeed, an optimal solution!

A very brief introduction to quantum computational complexity

So far, we have focused only on measuring time complexity with classical models. However, this is a book on quantum computing, so it is natural to ask what will change if we consider quantum computational models instead. This is studied in **quantum computational complexity theory**, a fascinating topic that is totally beyond the scope of this book.

Let us, however, say a few words on the kind of concepts that arise when quantum models are considered instead of classical Turing machines. This is not at all needed to understand any other part of the book, so feel completely free to skip it. We will need to be brief, but you can refer to the survey by Watrous [110] for more details.

It turns out that it is possible to define a class of problems that can be seen as a quantum analogous to P. This class is known as *BQP*, and it contains those decision problems that can be solved with bounded error in polynomial time with a quantum algorithm.

There are a couple of things that we need to clarify here. The first one is that quantum algorithms being probabilistic, we cannot expect the correct answer to a decision problem to always be obtained. Instead, we impose that this correct answer is returned, for each input, with high probability. Formally, the requirement is that for every positive instance x, the probability of obtaining 1 when the input to the algorithm is x should be at least $2/3$; similarly, for every negative instance x, the probability of obtaining 0 when the algorithm runs on x should be at least $2/3$. In this way, we can repeat the procedure with the same input several times and take the majority result. If the number of repetitions is big enough (but fixed), we can make the probability of error arbitrarily small while still having a total running time that is polynomial.

To learn more…

BQP is not exactly analogous to P but to another (classical) computational class called BPP. The class BPP contains those decision problems that can be solved with bounded error in polynomial time with a probabilistic Turing machine (that is, a Turing machine with multiple instructions for certain state-symbol situations and that can decide which instruction to execute based on a sequence of random bits). BPP stands for **bounded-error probabilistic polynomial time** while BQP stands for **bounded-error quantum polynomial time**.

The other thing that needs to be clarified about our definition of BQP is what we exactly understand by a quantum algorithm. In the classical case, we have identified this notion with a (single-tape) Turing machine. It is possible to define a quantum version of Turing machines (see, for instance, the paper by Bernstein and Vazirani [111]) and use it in our definition. But since our primary model for quantum computations throughout this book is the quantum circuit model, a natural question is whether we can also use it to formalize the notion of quantum algorithm.

In fact, we can give a definition of what is a quantum algorithm in terms of quantum circuits, and this definition is equivalent in computational power to the one in terms of quantum Turing machines (and polynomially equivalent with respect to running time). However, there exist several subtleties that need to be confronted.

The first one is related to being able to consistently measure the execution time of a quantum circuit. To do that, we need to fix a finite set of gates and express every circuit using only those gates. Then, we can assign a cost of one unit to each of those gates and measure the running time of a circuit as its total number of gates. Otherwise, if we allow arbitrary gates, then we could argue that any circuit is just a single unitary gate (plus some measurements), something that is clearly meaningless in terms of analyzing its complexity. Notice that fixing a finite set of permitted gates also allows us to describe every circuit as a finite binary string, for instance, giving a list of the gates that we use and the qubits on which we apply them.

The finite set of gates needs to be chosen in a way that we can approximate any given quantum circuit to arbitrary precision. A possible way of doing this is explained in the survey by Watrous [110].

A second technical problem that we need to tackle is that, while a Turing machine can process inputs of any size, every quantum circuit has a fixed number of qubits and, hence, only admits inputs of a fixed size. As a consequence, we cannot represent a full algorithm (that needs to be able to solve every possible instance of a problem) with just one quantum circuit: we need to consider an infinite family of circuits, one for each input size. So, a quantum algorithm is not a single quantum circuit, but a collection $\{C_n\}$ of circuits, one for each natural number n, so that C_n admits n qubits as its input.

The final issue that we need to address is related to the way in which we select that infinite family of circuits. If we allow any collection of circuits to represent a quantum algorithm, then we can end up in pathological situations such as being able to solve (a problem equivalent to) the Halting problem, which we know to be uncomputable! This is because we could just select a different, totally unrelated quantum circuit for each size in a way that the quantum circuit already "knows" the answer to the Halting problem for its input size. This is not something particular to just quantum circuits. The same happens with classical Boolean circuits (as we mentioned, this is a subtle point; see Section 2.2 in the book by Kitaev et al. [112] or Chapter 6 in the book by Arora and Barak [103], especially what is said there about the $P/poly$ class of problems).

The solution to this issue is to specify all the quantum circuits in the family in a **uniform** way. For instance, we can impose that there exists a (classical) Turing machine that, given a natural number n, generates the circuit for input size n in polynomial time (in n). In this way, we can't hide any additional complexity in the selection of the quantum circuits. Remember that we can represent our quantum circuits as finite binary strings (because we have fixed a finite number of allowable quantum gates), so it makes sense to obtain them as the output of a Turing machine. Moreover, every circuit will have a polynomial size (a polynomial-time Turing machine can only output a polynomial number of bits, after all) and hence a polynomial running time.

Important note

BQP is the class of decision problems that can be solved with bounded error by polynomial-time uniform families of quantum circuits.

Now that we have defined BQP, it is natural to ask about its relationship with P and NP in order to be able to assess the power of quantum computers when compared to that of classical ones.

It is easy to show that $P \subseteq BQP$, that is, that every problem in P is also in BQP. This follows directly from the fact that we can simulate any classical Boolean circuit with a quantum circuit (as we show in *Section 1.5.2*) and from the fact that polynomial-time uniform families of classical circuits are equivalent to polynomial-time Turing machines (see Section 6.2 in the book by Arora and Barak [103]). But this is not surprising at all, because we expect quantum computers to be at least as powerful as classical computers.

So the question that we should really ask is whether there are problems in BQP that are not in P. The short answer is that…we don't know. Proving it would imply a major breakthrough not only in quantum computational complexity but also in classical computational complexity theory. It can be proved that BQP is contained in $PSPACE$, the class of decision problems solvable in polynomial space. Showing that P is different from BQP would also imply that P is different from $PSPACE$, which is a major open question in computational complexity (although it should be easier to solve than the P versus NP problem, because NP is also contained in $PSPACE$).

That being said, we have good reasons to believe that there are problems in *BQP* that are not in *P*. In fact, we have a very good candidate: the factoring problem (given natural numbers m and k, check whether m has a factor $l \neq 1$ that is less than k) is in *BQP* thanks to Shor's algorithm [6], but it would be really, really surprising if it were in *P*. In fact, many cryptographic protocols currently in use rely on the assumption that factoring is not in *P*. So, every time that you buy something online and you send your credit card number over the internet, you are implicitly trusting that *P* and *BQP* are not equal (and that nobody owns a powerful enough quantum computer!).

And what about *BQP* and *NP*? The situation there is a little bit more complicated. The evidence that we have seems to imply that there are problems in *BQP* that are not in *NP* (one of the strongest results in this direction can be found in a recent paper by Raz and Tal [113]). But we also have some evidence that seems to suggest that there are problems in *NP* that are not in *BQP*, due to results by Bennett, Bernstein, Brassard, and Vazirani [114] that show that Grover's algorithm is, in a certain sense, optimal among quantum algorithms for search tasks.

If all this is true, it would imply that there are problems that we can solve efficiently with quantum algorithms that we couldn't solve efficiently even with non-deterministic machines. But, contrary to what can be read sometimes in the media, it also would imply that not every problem in *NP* could be solved efficiently with a quantum computer, even if it were fault-tolerant. In particular, it would imply that no *NP*-complete problem could be solved efficiently with quantum algorithms (we have represented all these relationships in *Figure C.2*).

Does this mean that quantum computers are not useful at all for optimization problems? Not necessarily. The methods that we describe in *Part 2* of this book may not be able to give the optimal solution to every optimization problem out there. But they provide approximation algorithms that might beat whatever is possible with just classical algorithms. For instance, the QAOA algorithm that we study in *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*, is considered a possible candidate for that kind of advantage (for some recent results in this direction, see the papers by Basso et al. [115] and by Farhi et al. [116], but

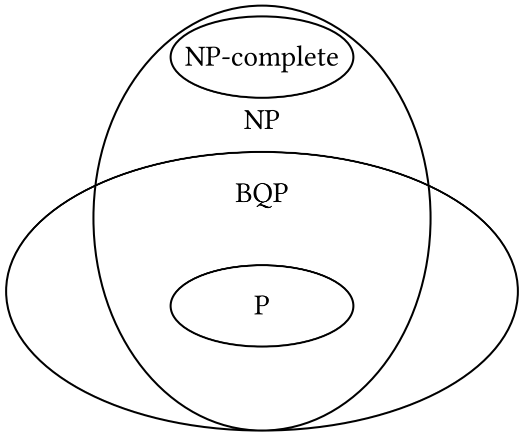

Figure C.2: Possible relationships between P, NP, BQP, and NP-complete problems according to the available evidence and the most accepted conjectures. Be warned: some of these classes might end up being completely equal!

also check the response by Hastings [117]). And even if that were not the case, methods such as quantum annealing (described in *Chapter 4, Adiabatic Quantum Computing and Quantum Annealing*) or QAOA may provide good heuristics that are useful in practice, in the same way that genetic algorithms, simulated annealing, or particle-swarm optimization are used to solve practical problems in many different fields.