Chapter 4
Quantum Universality, Computability, & Complexity

"[. . .] *Turing's theory is not entirely mathematical* [. . .]. *It makes hidden assumptions about physics which are not quite true. Turing and other physicists who constructed universal models for classical computation tried hard not to make any assumptions about the underlying physics* [. . .]. *But their intuition did not encompass quantum theory, and their imaginary paper did not exhibit quantum coherence*."
– David Deutsch[1]

Once while visiting Stephen Hawking in Cambridge, England, Stephen asked me what I was working on. At the time I was a research scientist at Xerox PARC developing what later became called the theory of computational phase transitions, which is a view of computation inspired by statistical physics that I will describe in Chap. 7. However, since the term "computational phase transition" was generally unknown at that time, I replied by saying I was working on "computational complexity theory". I distinctly recall an expression of disdain sweep across Stephen's face, and the conversation quickly switching to something else. In retrospect, Stephen's pained expression turned out to be prophetic for many subsequent conversations I have had with other physicists. It appears physicists are not generally enamored with computational complexity theory!

Why is this? In part, I believe it is a cultural difference. I have found that physicists tend to embrace simplified approximate models that encourage comprehension, whereas computer scientists tend to prefer detailed exact models about which strong theorems can be proved. Neither style is right nor wrong—just different. Moreover, physicists have an uncanny knack for picking terminology that is vivid, and alluring, e.g., "Big Bang", "dark matter", "black hole", "twin-paradox", "strange attractor" etc., whereas theoretical computer science is replete with the most über-geeky nomenclature imaginable as exemplified by the byzantine names of computational complexity classes. My complaint is not so much about the archaic names theoretical computer scientists have chosen, but the ad hoc ways in which the system of names has been expanded. Had we done the same with organic chemistry key

[1] Source: in David Deutsch, "Quantum Computation," Physics World, June (1992) pp. 57–61.

C.P. Williams, *Explorations in Quantum Computing,*
Texts in Computer Science,
DOI 10.1007/978-1-84628-887-6_4,

insights and generalizations might have been missed. Had we picked a more systematic naming convention that aids comprehension of the concepts underpinning the complexity classes and how they differ from one another, then perhaps greater insights, or more useful classes, might have been discovered. The current nomenclature does not, in my opinion, assist comprehension of the underlying complexity class distinctions and their interrelationships.

Despite these differences, both fields have revealed extremely counter-intuitive, intriguing, and profound results. In this chapter, we highlight some of these amazing results from theoretical computer science and ask whether or not they still hold true in the quantum domain.

First, there is the question of *complexity*: Can a quantum computer perform the same tasks as a classical computer, but in significantly fewer steps? Second, there is the question of *computability*; Can a quantum computer *perform* computations that a classical computer cannot? And finally there is the question of *universality*; Is there a specialized quantum computer that can simulate any other quantum computer, and classical computer, *efficiently*? A difference between the capabilities of a quantum computer and those of a classical computer on any one of these criteria would be significant.

4.1 Models of Computation

To answer questions about complexity, universality, and computability, one must have a model of computation in mind. In the 1930's three superficially different models of computation were invented by Alan Turing, Emil Post, Kurt Gödel and Alonzo Church.

4.1.1 The Inspiration Behind Turing's Model of Computation: The Entscheidungsproblem

In 1900, Hilbert gave an address at the International Congress of Mathematics held in Paris concerning what he believed to be the 23 most challenging mathematical problems of his day. The last problem on his list asked whether there was a mechanical procedure by which the truth or falsity of any mathematical conjecture could be decided. In German, the word for "decision" is "entscheidung," so Hilbert's 23rd problem became known as the *"Entscheidungsproblem"*. Turing's abstract model of computation grew out of his attempt to answer the Entscheidungsproblem.

Hilbert's motivation for asking this question arose from the trend towards abstraction in mathematics. Throughout the 19th century, mathematics was largely a practical matter, concerned with making statements about real-world objects. In the late 1800s mathematicians began to invent, and then reason about, imaginary objects to which they ascribed properties that were not necessarily compatible with

"common sense." Thus the truth or falsity of statements made about such imaginary objects could not be determined by appealing to the real world. In an attempt to put mathematical reasoning on secure logical foundations, Hilbert advocated a "formalist" approach to proofs. To a formalist, symbols cease to have any meaning other than that implied by their relationships to one another. No inference is permitted unless there is an explicit rule that sanctions it, and no information about the meaning of any symbol enters into a proof from outside itself. Thus the very philosophy of mathematics that Hilbert advocated seemed very machine-like, and hence Hilbert proposed the *Entscheidungsproblem*.

Turing heard about Hilbert's *Entscheidungsproblem* during a course of lectures, given by Max Newman, which he attended at Cambridge University. In his lecture Newman had described the Entscheidungsproblem as asking whether there was be a "mechanical" means of deciding the truth or falsity of a mathematical proposition. Although Newman probably meant "mechanical" figuratively, Turing interpreted it literally. Turing wondered whether a machine could exist that would be able to decide the truth or falsity of any mathematical proposition. Thus, in order to address the *Entscheidungsproblem*, Turing realized that he needed to model the process in which a human mathematician engages when attempting to prove some mathematical conjecture.

Mathematical reasoning is an enigmatic activity. We do not really know what goes on inside a mathematician's head, but we can examine the result of his thought processes in the form of the notes he creates whilst developing a proof. Mathematical reasoning consists of combining axioms (statements taken to be true without proof) with rules of logical inference, to infer consequents, which themselves become additional nuggets of information upon which further inferences may be drawn. So the reasoning process builds on itself and will result in valid conclusions provided the starting axioms are correct and the rules of inference are valid.

Turing abstracted the process followed by the mathematician into four principal ingredients: a set of transformation rules that allowed one mathematical statement to be transformed into another; a method for recording each step in the proof, an ability to go back and forth over the proof to combine earlier inferences with later ones, and a mechanism for deciding which rule to apply at any given moment. This is the essence of the proof process (at least its visible part). Next, Turing sought to simplify these steps in such a way that a machine could be made to imitate them. Mathematical statements are built up out of a mixture of ordinary letters, numbers, parentheses, operators (e.g., plus, "$+$" and times "$\times$") and special mathematical symbols (e.g., $\forall, \exists, \neg, \wedge, \vee$). Turing realized that the symbols themselves were of no particular significance. All that mattered was that they were used consistently and that their number was finite. Moreover, once you know you are dealing with a finite alphabet, you can place each symbol in one-to-one correspondence with a unique pattern of any two symbols (such as 0 and 1). Hence, rather than deal with a rich array of esoteric symbols, Turing realized that a machine only needed to be able to read and write two kinds of symbol, 0 and 1, say, with blank spaces or some other convention to identify the boundaries between the distinct symbols. Similarly, the fact that the scratch pad on which the mathematician writes intermediate results

is two-dimensional is of no particular importance. You could imagine attaching the beginning of one line of a proof to end of the previous line, making one long continuous strip of paper. So, for simplicity, Turing assumed that the proof could be written out on a long strip of paper or a "tape." Moreover, rather than allowing freeform handwriting, it would clearly be easier for a machine to deal with a tape marked off into a sequence of identical cells and only permitting one symbol to be written inside each cell, or the cell to be left blank.

Finally, the process of the mathematician going back and forth over previous conclusions in order to draw new ones could be captured by imagining that there is a "read/write" head going back and forth along the tape. When a mathematician views an earlier result it is usually in some context. A mathematician might read a set of symbols, write something, but come back to read those same symbols again later, and write something else. Thus, the context in which a set of symbols is read can affect the subsequent actions. Turing captured this idea by defining the "head" of his Turing machine to be in certain "states," corresponding to particular contexts. The combination of the symbol being read under the head and the state of the machine determined what symbol to write on the tape, which direction to move the head, and which state to enter next.

This is clearly a crude model of the proof process. Nevertheless it turned out to be surprisingly powerful. No matter what embellishments people dreamed up, Turing could always argue that they merely were refinements to some existing part of the model rather than being fundamentally new features. Consequently the Turing machine model was indeed the essence of the proof process. By putting the aforementioned mechanistic analogues of human behavior into a mathematical form, Turing was led to the idea of a "deterministic Turing machine".

4.1.2 Deterministic Turing Machines

The most influential model of computation was invented by Alan Turing in 1936 [501]. A Turing machine is an idealized mathematical model of a computer that can be used to understand the limits of what computers can do [237]. It is not meant to be a practical design for any actual machine but rather a simplified abstraction that, nevertheless, captures the essential features of any real computer. A Turing machine's usefulness stems from being sufficiently simple to allow mathematicians to prove theorems about its computational capabilities and yet sufficiently complex to accommodate any actual classical digital computer, no matter how it is implemented in the physical world.

A deterministic Turing machine is illustrated in Fig. 4.1. Its components are inspired by Turing's abstract view mathematical reasoning. A deterministic Turing machine consists of an infinitely long tape that is marked off into a sequence of cells on which may be written a 0 or a 1, and a read/write head that can move back and forth along the tape scanning the contents of each cell. The head can exist in one of a finite set of internal "states" and contains a set of instructions (constituting

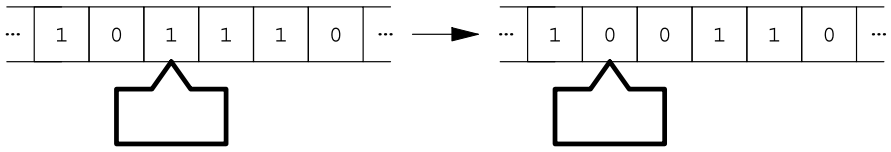

Fig. 4.1 A deterministic Turing machine

the "program") that specifies, given the current internal state, how the state must change given the bit (i.e., the binary digit 0 or 1) currently being read under the head, whether that bit should be changed, and in which direction the head should then be advanced.

The tape is initially set up in some standardized state such as all cells containing 0 except for a few that hold the program and any initial data. Thereafter the tape serves as the scratch pad on which all intermediate results and the final answer (if any) are written.

Despite its simplicity, the Turing Machine model has proven to be remarkably durable. In the 70-odd years since its inception, computer technology has advanced considerably. Nevertheless, the Turing machine model remains as applicable today as it was back in 1936. Although we are apt to think of multimillion dollar supercomputers as being more powerful than humble desktop machines, the Turing machine model proves otherwise. Given enough time and memory capacity there is not a single computation that a supercomputer can perform that a personal computer cannot also perform. In the strict theoretical sense, they are equivalent. Thus the Turing machine is the foundational upon which much of current computer science rests. It has enabled computer scientists to prove many theorems that bound the capabilities of computing machinery.

More recently, however, a new idea has emerged that adds a slight twist to the deterministic Turing machine. Deterministic Turing machines, which follow rigid pre-defined rules, are susceptible to systematic biases that can cause them to take a very long time to solve certain problems. These are the problems for which the particular set of deterministic rules happen to make the Turing machine examine almost all the potential solutions before discover an actual solution. For example, if an adversary knew the rules by which a give DTM operated they could devise a problem that was guarantee to tax the machine to its maximum before finding a true solution. To avoid such pitfalls, a new type of Turing machine was invented that employs randomness, this is called a probabilistic, or non-deterministic, Turing machine.

4.1.3 Probabilistic Turing Machines

An alternative model of classical computation is to equip a deterministic Turing machine with the ability to make a random choice, such as flipping a coin. The result is a probabilistic Turing machine. Surprisingly, many problems that take a

long time to solve on a deterministic Turing machine (DTM) can often be solved very quickly on a probabilistic Turing machine (PTM).

In the probabilistic model of computation there are often tradeoffs between the time it takes to return an answer to a computation and the probability that the answer returned is correct. For example, suppose you wanted to plan a round the world trip that visited 100 cities, but you wanted to minimize the distance you have to travel between cities and you only wanted to visit each city once. The problem of computing the optimal (shortest path) route for your trip is extremely demanding computationally. However, if you were prepared to accept a route that was guaranteed to be only a little bit longer than the optimal route, and could in fact be the optimal route, then this problem is very easy to solve computationally. For example, the Euclidean TSP is known to be an **NP-Complete** problem [377], which means that, to the best knowledge of computer scientists at the present time, the computational cost of finding the *optimal* tour scales exponentially with the number of cities to be visited, N, making the problem intractable for sufficiently large N. Nevertheless, there is a randomized algorithm that can find a tour to within $\mathcal{O}(1 + 1/c)$ of the optimal tour (for any constant c) in a time that scales only as $\mathcal{O}(N(\log(N))^{O(c)})$ [20], which is worse than linear but better than exponential scaling. Thus, randomization can be a powerful tool for rendering intractable problems tractable provided we are content with finding a good approximation to the optimal or exact solution.

An alternative tradeoff, if you *require* a correct answer, is to allow uncertainty in the length of time the probabilistic algorithm must run before it returns an answer. Consequently, a new issue enters the computational theory, namely, the correctness of an answer and its relationship to the running time of an algorithm.

Whereas a deterministic Turing Machine, in a certain state, reading a certain symbol, has precisely one successor state available to it, the probabilistic Turing machine has multiple legitimate successor states available, as shown in Fig. 4.2. The choice of which state is the one ultimately explored is determined by the outcome of a random choice (possibly with a bias in favor of some states over others). In all other respects the PTM is just like a DTM. Despite the superficial difference between PTMs and DTMs, computer scientists have proved that anything computable by a probabilistic Turing machine can also be computed by a deterministic Turing machine, although in such cases the probabilistic machine is often more efficient [198]. The basic reason for the success of probabilistic approach is that a probabilistic algorithm can be thought of as swapping between a collection of deterministic algorithms. Whereas it is fairly easy to design a problem so that it will mislead a particular deterministic algorithm, it is much harder to do so for a probabilistic algorithm because it keeps on changing its "identity." Indeed the latest algorithms for solving hard computational problems now interleave deterministic, with probabilistic steps. The exact proportions of each strategy can have a huge impact on the overall efficiency of problem solving.

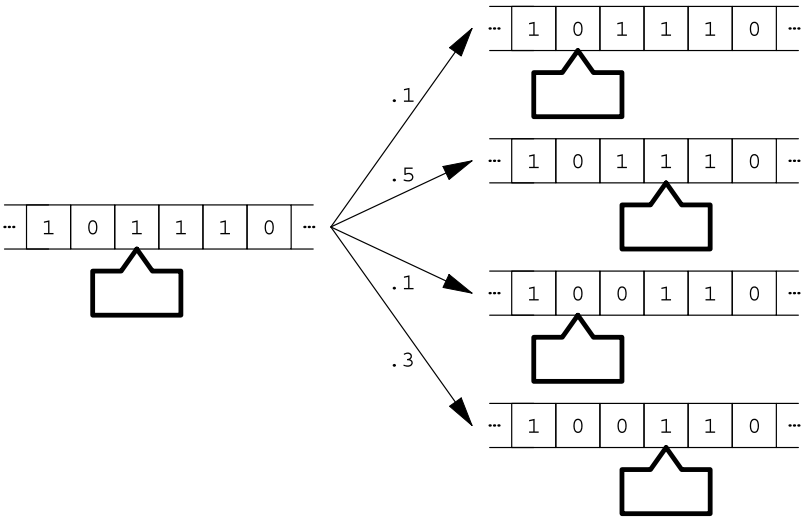

Fig. 4.2 In a probabilistic classical Turing machine there are multiple possible successor states, only one of which is actually selected and pursued at any one time

4.1.4 The Alternative Gödel, Church, and Post Models

Kurt Gödel invented a very different model of computation than that formulated by Turing. Gödel identified the tasks that a computer can perform with a class of *recursive functions*, i.e., functions that refer to themselves. For example, the function $\text{fib}(x) = \text{fib}(x-1) + \text{fib}(x-2)$ such that $\text{fib}(1) = \text{fib}(2) = 1$ defines a recursive function that generates the Fibonnaci sequence, i.e., as x takes on integer values $x = 1, 2, 3, 4, 5, 6, \ldots$, then $f(x)$ generates the Fibonnaci numbers $1, 1, 2, 3, 5, 8, \ldots$. The function $\text{fib}(\cdot)$ is defined in terms of itself, and is therefore a recursive function.

Yet another model of computation was formulated by Alonzo Church. Church equated the tasks that a computer can perform with the so-called λ-definable functions (which you will have encountered if you have ever used the LISP programming language). This viewed computation as a nesting of function evaluations. The simplicity of the λ-calculus made it possible to prove various properties of computations.

Hence both Gödel's and Church's formulations of computation viewed it as an elaborate mathematical function evaluation in which simpler functions were composed to make more elaborate ones.

Emil Post anticipated many of the results of Gödel, Turing, and Church but chose not publish them. His "finite combinatory processes—Formulation I" [397] is similar in spirit to the idea of a Turing machine. Post did not ever speak overtly of computing machines, but he did invent (independently of Turing) the idea of a human worker moving along a two way infinite "workspace" of boxes each of which could be marked or unmarked, and following a set of directions: a conditional jump, "Stop", move left, move right, mark box or unmark box.

4.1.5 Equivalence of the Models of Computation

Thus, Turing identified the tasks a computer can perform with the class of functions computable by a hypothetical computing device called a *Turing Machine*. This viewed computation a rather imperative or "procedural" style. Slightly later Emil Post also formalized computation in a similar machine model, which he asserted was "logically equivalent to recursiveness". Kurt Gödel equated computation with recursive functions and Alonzo Church with λ-definable functions.

Although, superficially, the models of computation advanced by Turing, Gödel, Church, and Post look different, it turns out that they are *equivalent* to one another. This was something of a surprise as there was no reason to expect their equivalence a priori.

Moreover, any one of the models alone might be open to the criticism that it provided an incomplete account of computation. But the fact that three radically different views of computation all turned out to be equivalent was a clear indication that the most important aspects of computation had been characterized correctly.

4.2 Universality

In the 1930s computer science was a rather fledgling field. People dabbled with building computers but very few machines actually existed. Those that did had been tailor-made for specific applications. However, the concept of a Turing machine raised new possibilities. Turing realized that one could encode the transformation rules of any particular Turing machine, T say, as some pattern of 0s and 1s on the tape that is fed into some special Turing machine, called U. U had the effect of reading in the pattern specifying the transformation rules for T and thereafter treated any further input bits exactly as T would have done. Thus U was a universal mimic of T and hence was called the *Universal Turing Machine*. Thus, one Turing machine could mimic the behavior of another.

4.2.1 The Strong Church-Turing Thesis

The ability to prove that all the competing models of classical computation were equivalent led Church to propose the following principle, which has subsequently become known as the Church-Turing thesis [450]:

Strong Church-Turing Thesis *Any process that is effective or algorithmic in nature defines a mathematical function belonging to a specific well-defined class, known variously as the recursive, the λ-definable, or the Turing computable functions. Of, in Turing's words, every function which would naturally be regarded as computable can be computed by the universal Turing machine.*

Thus a model of computation is deemed *universal*, with respect to a family of alternative models of computation, if it can compute any function computable by those other models either directly or via emulation.

4.2.2 Quantum Challenge to the Strong Church-Turing Thesis

Notwithstanding these successes, in the early 1980s a few maverick scientists began to question the correctness of the classical models of computation. The deterministic Turing machine and probabilistic Turing machine models are certainly fine as *mathematical* abstractions but are they consistent with known *physics*? This question was irrelevant in Turing's era because computers operated at a scale well above that of quantum systems. However, as miniaturization progresses, it is reasonable, in fact, necessary, to re-consider the foundations of computer science in the light of our improved understanding of the microscopic world.

Unfortunately, we now know that although these models were intended to be mathematical abstractions of computation that were free of physical assumptions, they do, in fact, harbor implicit assumptions about the physical phenomena available to a computer. These assumptions appear to be perfectly valid in the world we see around us, but they cease to be valid on sufficiently small scales.

We now know that the Turing Machine model contains a fatal flaw. In spite of Turing's best efforts, some remnants of classical physics, such as the assumption that a bit must be either a 0 *or* a 1, crept into the Turing machine models. The obvious advances in technology, such as more memory, more instructions per second, greater energy efficiency have all been merely quantitative in nature. The underlying foundations of computer science have not changed. Similarly, although certainly having a huge social impact, apparent revolutions, such as the explosion of the Internet, have merely provided new conduits for information to be exchanged. They have not altered the fundamental capabilities of computers in any way whatsoever. However, as computers become smaller, eventually their behavior *must* be described using the physics appropriate for small scales, that is, quantum physics.

The apparent discrepancy between Feynman's observation that classical computers cannot simulate quantum system efficiently and the Church-Turing thesis means that the Strong Church-Turing Thesis may be flawed for there is no known way to simulate quantum physics efficiently on any kind of classical Turing machine. This realization led David Deutsch in 1985 to propose reformulating the Church-Turing thesis in physical terms. Thus Deutsch prefers:

Deutsch's Thesis *Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.*

This can only be made compatible with Feynman's observation on the efficiency of simulating quantum systems by basing the universal model computing machine on quantum mechanics itself. This insight was the inspiration that allowed David

Deustch to prove that it was possible to devise a "Universal Quantum Turing Machine", i.e., a quantum Turing machine that could simulate any other quantum Turing machine. The efficiency of Deustch's Universal Quantum Turing Machine has since been improved upon by several other scientists.

We don't yet know how history with rate the relative contributions of various scientists to the field of quantum computing. Curiously though, if you search for "quantum computing" at www.wikiquote.com you will discover "David Deutsch, Relevance: 4.2%; Richard Feynman, Relevance: 2.2% and (my personal favorite) God, Relevance: 0.9%". I have to say that I think wikiquote has it about right! I certainly concur with the relative ratings of Deutsch's and Feynman's contributions, but I will leave it to each author (one living, one dead) to argue with the Almighty Himself, the merits of their ranking with respect to God.

4.2.3 Quantum Turing Machines

The first quantum mechanical description of a Turing machine was given by Paul Benioff in 1980 [43]. Benioff was building on earlier work carried out by Charles Bennett who had shown that a reversible Turing machine was a theoretical possibility [44].

A reversible Turing machine is a special version of a deterministic Turing machine that never erases any information. This is important because physicists had shown that, in principle, all of the energy expended in performing a computation can be recovered provided that the computer does not throw any information away. The notion of "throwing information away" means that the output from each step of the machine must contain within it enough information that the step can be undone without ambiguity. Thus, if you think of a reversible Turing machine as a dynamical system, then given knowledge of its state at any one moment would allow you to predict its state at all future and all past times. No information was ever lost and the entire computation could be run forwards or backwards.

This fact struck a chord with Benioff, for he realized that any isolated quantum system had a dynamical evolution that was reversible in exactly this sense. Thus it ought to be possible to devise a quantum system whose evolution over time mimicked the actions of a classical reversible Turing machine. This is exactly what Benioff did. Unfortunately, Benioff's machine is not a true quantum computer. Although between computational steps the machine exists in an intrinsically quantum state (in fact a "superposition," of computational basis states, at the end of each step the "tape" of the machine was always back in one of its classical states: a sequence of classical bits. Thus, Benioff's design could do no more than a classical reversible Turing machine.

The possibility that quantum mechanical effects might offer something genuinely new was first hinted at by Richard Feynman of Caltech in 1982, when he showed that no classical Turing machine could simulate certain quantum phenomena without incurring an unacceptably large slowdown but that a "universal quantum simulator"

could do so. Unfortunately, Feynman did not provide a design for such a simulator, so his idea had little immediate impact. Nor did he did not prove, conclusively, that a universal quantum simulator was possible. However, indeed it is. The question was answered in the affirmative by Seth Lloyd in 1996 [321].

The key step in making it possible to study the computational power of quantum computers came in 1985, when David Deutsch of Oxford University, described the first true quantum Turing machine (QTM) [136]. A QTM is a Turing machine whose read, write, and shift operations are accomplished by quantum mechanical interactions and whose "tape" can exist in states that are highly nonclassical. In particular, whereas a conventional classical Turing machine can only encode a 0, 1, or blank in each cell of the tape, the QTM can exist in a blend, or "superposition" of 0 and 1 simultaneously. Thus the QTM has the potential for encoding many inputs to a problem simultaneously on the same tape, and performing a calculation on all the inputs in the time it takes to do just one of the calculations classically. This results in a superposition of all the classical results and, with the appropriate measurement, you can extract information about certain joint properties of all these classical results. This technique is called "quantum parallelism." We saw an example of quantum parallelism when we solved Deutsch's problem in Chap. 1.

Moreover, the superposition state representing the tape of the QTM can correspond to an entanglement of several classical bit string configurations. Entanglement means that the quantum state of the entire tape is well-defined but the state of the individual qubits is not. For example, a 3-qubit tape in the state $\frac{1}{\sqrt{2}}(|010\rangle + |101\rangle)$ represents an entanglement of the two configurations $|010\rangle$ and $|101\rangle$. It is entangled in the sense that it you were to measure any one of these qubits, the quantum state of the other two qubits would become definite instantaneously. Thus, if you read out the bit values from a part of the tape of the QTM when it is in an entangled state, your actions will have a side effect on the state of the other (unmeasured) qubits. In fact it is the existence of such "entangled" qubits that is the fundamental reason QTMs are different from classical deterministic and probabilistic TMs.

A graphical representation of a QTM is shown in Fig. 4.3. There is a *single* physical tape running from left to right in the figure. However, this single tape is drawn as if it were several tapes in parallel to convey the idea that the single quantum tape can hold a superposition of many different bit strings simultaneously.

As we saw in Chap. 1, each qubit in a QTM, when considered is perfect isolation from other qubits, can be visualized as a small arrow contained in a sphere. "Straight up" represents the (classical) binary value 0 and "straight down" represents the (classical) binary value 1. When the arrow is at any other orientation, the angle the arrow makes with the horizontal axis is a measure of the ratio of 0-ness to 1-ness in the qubit. Likewise, the angle through which the arrow is rotated about the vertical axis is a measure of the "phase". Thus, drawing qubits as arrows contained in spheres we can depict a typical superposition state of Deutsch's quantum Turing machine as shown in Fig. 4.3. The possible successor states of the tape are indicated by edges between different possible tape configurations.

Quantum Turing machines (QTMs) are best thought of as quantum mechanical generalizations of probabilistic Turing machines (PTMs). In a PTM, if you initialize

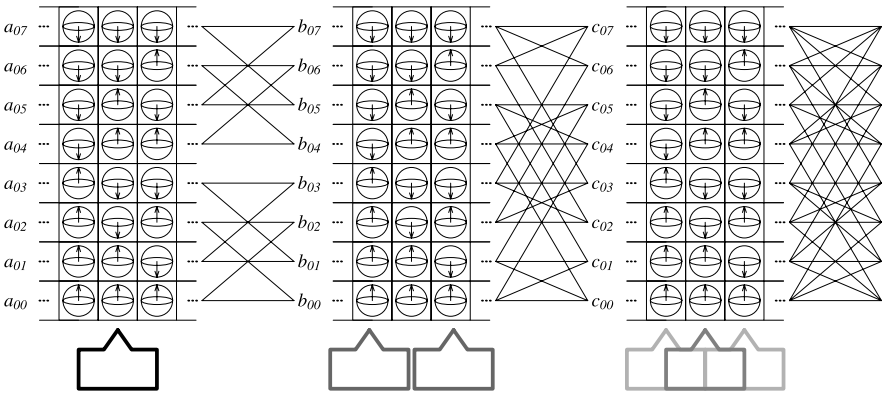

Fig. 4.3 In the quantum Turing machine, each cell on the tape can hold a qubit. In this figure there is *one* physical tape but it is drawn as multiple tapes corresponding to a different bit pattern for each component of the net superposition state

the tape in some starting configuration and run the machine without inspecting its state for t steps, then its final state will be uncertain and can only be described using a probability distribution over all the possible states accessible in t steps.

Likewise, in a QTM if you start the machine off in some initial configuration, and allow it to evolve for t steps, then its state will be described by a superposition of all states reachable in t steps. The key difference is that in a classical PTM only one particular computational trajectory is followed, but in the QTM *all* computational trajectories are followed and the resulting superposition is the sum over all possible states reachable in t steps. This makes the calculation of the net probability of a particular computational outcome different for a PTM than a QTM.

In the PTM if a particular answer can be reached independently, in more than one way, the net *probability* of that answer is given by the sum of each probability that leads to that answer. However, in the QTM if a given answer can be reached in more than one way the net probability of obtaining that answer is given by summing the *amplitudes* of all trajectories that lead to that answer and then computing their *absolute value squared* to obtain the corresponding probabilities.

If the quantum state of the QTM in Fig. 4.3 is the superposition $c_0|00000\rangle + c_1|00001\rangle + c_2|00010\rangle + \cdots + c_{31}|11111\rangle$ the coefficients $c_0, c_1, \ldots, c_{31}$ are the amplitudes, and probability of finding the tape of the QTM in the bit configuration $|00010\rangle$, say, when you read each of the bits is equal to $|c_2|^2$. If an event occurs with a probability of 0 this means that there is a 0% chance, i.e., utter impossibility, of that event occurring. Conversely, if an event occurs with a probability of 1 this means that there is a 100% chance, i.e., absolutely certainty, that the event will occur.

Whereas classical probabilities are real numbers between zero and one, "amplitudes" are complex numbers (i.e. numbers of the form $x + iy$ where x and y are real numbers). When you add two probabilities you always get a bigger or equal probability. However, when you add two complex amplitudes together they do not always result in a number that has a bigger absolute value. Some pairs of amplitudes

tend to cancel each other out resulting in a net reduction in the probability of seeing a particular outcome. Other pairs of amplitudes tend to reinforce one another and thereby enhance the probability of a particular outcome. This is the phenomenon of quantum interference.

Quantum interference is a very important mechanism in quantum computing. Typically, when designing a quantum computer to solve a hard computational problem, you have to devise a method (in the form of a quantum algorithm) to evolve a superposition of all the valid inputs to the problem into a superposition of all the valid solutions to that problem. If you can do so, when you read the final state of your memory register you will be guaranteed to obtain one of the valid solutions. Understanding how to achieve your desired evolution invariably entails arranging for the computational pathways that lead to non-solutions to interfere destructively with one another and hence cancel out, and arranging for the computational pathways that lead to solutions to interfere constructively and hence reinforce one another.

Armed with this model of an abstract quantum Turing machine, several researchers have been able to prove theorems about the capabilities of quantum computers [58]. This effort has focused primarily on *universality* (whether one machine can simulate all others efficiently), *computability* (what problems the machines can do), and *complexity* (how the memory, time and communication resources scale with problem size). Let us take a look at each of these concepts and compare the perspective given to us by classical computing and quantum computing.

4.3 Computability

Computability theory is concerned with which computational tasks, for a particular model of computation, can and cannot be accomplished within a finite length of time. If there is no algorithm, with respect to a particular model of computation, that can guarantee to find an answer to a given problem in a finite amount of time, that answer is said to be *uncomputable* with respect to that model of computation. One of the great breakthroughs in classical computer science was the recognition that all of the candidate models for computers, Turing machines, recursive functions, and λ-definable functions were equivalent in terms of what they could and could not compute. It is natural to wonder whether this equivalence extends to quantum computation too.

If you ask a young child what a computer can do you might be told, "They let me learn letters and numbers and play games." Ask a teenager and you might hear, "They let me surf the Web and meet online in chat rooms with my friends." Ask an adult and you might discover, "They're great for email, word processing and keeping track of my finances." What is remarkable is that the toddler, the teenager, the parent might all be talking about the *same* machine! By running the appropriate software it seems we can make the computer perform almost any task.

The possibility of one machine simulating another gave a theoretical justification for pursuing the idea of a programmable computer. In 1982, Richard Feynman observed that it did not appear possible for a Turing machine to simulate certain quan-

tum physical processes without incurring an exponential slowdown [181]. Here is an example.

Suppose you want to use a classical computer to simulate a quantum computer. Let's assume that the quantum computer is to contain n qubits and that each qubit is initially in a superposition state, $c_0|0\rangle + c_1|1\rangle$. Each such superposition is described by two complex numbers, c_0 and c_1, so we need a total of $2n$ complex numbers to describe the initial state of all n qubits when they are in this product state form.

Now what happens if we want to simulate a joint operation on all n qubits? Well, you'll find that the cost of the simulation skyrockets. Once we perform a joint operation on all n qubits, i.e., once we evolve them under the action of some quantum algorithm, they will most likely become entangled with one another. Whereas the initial state that we started with could be factored into a product of a state for each qubit, an entangled state cannot be factored in this manner. In fact, to even write down an arbitrary entangled state of n qubits requires 2^n complex numbers. Thus, as a classical computer must keep track of all these complex numbers explicitly, the cost of a classical simulation of a quantum system requires a huge amount of memory and computer time.

What about a quantum computer? Could a quantum computer simulate any quantum system efficiently? There is a good chance that it could because the quantum computer would have access to exactly the same physical phenomena as the system it is simulating. This result poses something of a problem for traditional (classical) computer science.

4.3.1 Does Quantum Computability Offer Anything New?

Is it possible to make more pointed statements about computability and quantum computers?

The first work in this area appeared in David Deutsch's original paper on quantum Turing machines [136]. Deutsch argued that quantum computers could compute certain outputs, such as true random numbers, that are not computable by any deterministic Turing machine. Classical deterministic Turing machines can only compute functions, that is, mathematical procedures that return a single, reproducible, answer. However, there are certain computational tasks that cannot be performed by evaluating any function. For example, there is no *function* that generates a *true* random number. Consequently, a Turing machine can only feign the generation of random numbers.

In the same paper, Deutsch introduced the idea of quantum parallelism. Quantum parallelism refers to the process of evaluating a function once on a blend or "superposition" of all possible inputs to the function to produce a superposition of outputs. Thus all the outputs are computed in the time taken to evaluate just one output classically. Unfortunately, you cannot obtain all of these outputs explicitly because a measurement of the final superposed state would yield only one output. Nevertheless, it is possible to obtain certain joint properties of all of the outputs.

In 1991 Richard Jozsa gave a mathematical characterization of the class of functions (i.e., joint properties) that were computable by quantum parallelism [261]. He discovered that if f is some function that takes integer arguments in the range 1 to m and returns a binary value, and if the joint property function J that defines some collective attribute of all the outputs of f, takes m binary values and returns a single binary value, then only a fraction $(2^{2^m} - 2^{m+1})/(2^{2^m})$ of all possible joint property functions are computable by quantum parallelism.

Thus quantum parallelism alone is not going to be sufficient to solve all the joint property questions we might wish to ask. Of course, you could always make a QTM simulate a classical TM and compute a particular joint property in that way. Although this is feasible, it is not desirable, because the resulting computation would be no more efficient on the quantum computer than on the classical machine. However, the ability of a QTM to simulate a TM means that the class of functions computable on QTMs exactly matches the class of functions computable on classical TMs.

4.3.2 Decidability: Resolution of the Entscheidungsproblem

It was, you will recall, a particular question regarding computability that was the impetus behind the Turing machine idea. Hilbert's *Entscheidungsproblem* had asked whether there was a mechanical procedure for deciding the truth or falsity of any mathematical conjecture, and the Turing machine model was invented to prove that there was no such procedure.

To construct this proof, Turing used a technique called *reductio ad absurdum*, in which you begin by assuming the truth of the opposite of what you want to prove and then derive a logical contradiction. The fact that your one assumption coupled with purely logical reasoning leads to a contradiction proves that the assumption must be faulty. In this case the assumption is that there *is* a procedure for deciding the truth or falsity of any mathematical proposition and so showing that this leads to a contradiction allows you to infer that there is, in fact, no such procedure.

The proof goes as follows: if there *were* such a procedure, and it were truly mechanical, it could be executed by some Turing machine with an appropriate table of instructions. But a "table of instructions" could always be converted into some finite sequence of 1s and 0s. Consequently, such tables can be placed in an order, which meant that the things these tables represented (i.e., the Turing machines) could also be placed in an order.

Similarly, the statement of any mathematical proposition could also be converted into a finite sequence of 1s and 0s; so they too could be placed in an order. Hence Turing conceived of building a table whose vertical axis enumerated every possible Turing machine and whose horizontal axis, every possible input to a Turing machine.

But how would a machine convey its decision on the veracity of a particular input, that is, a particular mathematical proposition? You could simply have the machine

Table 4.1 Turing's Table. The i-th row is the sequence of outputs of the i-th Turing machine acting on inputs 0, 1, 2, 3, ...

i-th DTM	j-th Input							
	0	1	2	3	4	5	6	...
0	⊗	⊗	⊗	⊗	⊗	⊗	⊗	...
1	0	0	0	0	0	0	0	...
2	1	2	1	⊗	3	0	⊗	...
3	2	0	0	1	5	7	⊗	...
4	3	⊗	1	8	1	6	9	...
5	7	1	⊗	⊗	5	0	0	...
6	⊗	2	4	1	7	3	4	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Table 4.2 Turing's Table after diagonal slash

i-th DTM	j-th Input							
	0	1	2	3	4	5	6	...
0	0	0	0	0	0	0	0	...
1	0	0	0	0	0	0	0	...
2	1	2	1	0	3	0	0	...
3	2	0	0	1	5	7	0	...
4	3	0	1	8	1	6	9	...
5	7	1	0	0	5	0	0	...
6	0	2	4	1	7	3	4	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

print out the result and halt. Hence the *Entscheidungsproblem* could be couched as the problem of deciding whether the i-th Turing machine acting on the j-th input would ever halt. Thus Hilbert's *Entscheidungsproblem* had been refashioned into Turing's Halting Problem.

Turing wanted to prove that there was no procedure by which the truth or falsity of a mathematical proposition could be decided; thus his proof begins by assuming the opposite, namely, that there *is* such a procedure. Under this assumption, Turing constructed a table whose (i, j)-th entry was the output of the i-th Turing machine on the j-th input, if and only if the machine halted on that input, or else some special symbol, such as ⊗, signifying that the corresponding Turing machine did not halt on that input. Such a table would resemble that shown in Table 4.1.

Next Turing replaced each symbol ⊗ with the bit "0". The result is shown in Table 4.2: Now because the rows enumerate all possible Turing machines and the columns enumerate all possible inputs (or, equivalently, mathematical propositions) all possible sequences of outputs, that is, all computable sequences of 1s and 0s,

Table 4.3 Turing's Table with 1 added to each element on the diagonal slash

i-th DTM	j-th Input							
	0	1	2	3	4	5	6	…
0	1	0	0	0	0	0	0	…
1	0	1	0	0	0	0	0	…
2	1	2	2	0	3	0	0	…
3	2	0	0	2	5	7	0	…
4	3	0	1	8	2	6	9	…
5	7	1	0	0	5	1	0	…
6	0	2	4	1	7	3	5	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

must be contained somewhere in this table. However, since any particular output is merely some sequence of 1s and 0s it is possible to change each one in some systematic way, for example by flipping one of the bits in the sequence. Consider incrementing each element on a diagonal slash through the table as shown in Table 4.3. The sequence of outputs along the diagonal differs in the i-th position from the sequence generated by the i-th Turing machine acting on the i-th input. Hence this sequence cannot appear in any of the rows in the table. However, by construction, the infinite table is supposed to contain *all* computable sequences and yet here is a sequence that we can clearly compute and yet cannot appear in any one row! Hence Turing established a contradiction and the assumption underpinning the argument must be wrong. That assumption was "there exists a procedure that can decide whether a given Turing machine acting on a given input will halt." As Turing showed that the Halting problem was equivalent to the Entscheidungsproblem, the impossibility of determining whether a given Turing machine will halt before running it shows that the Entscheidungsproblem must be answered in the negative too. In other words, there is no procedure for deciding the truth or falsity of all mathematical conjectures.

4.3.3 Proof Versus Truth: Gödel's Incompleteness Theorem

In 1936 Kurt Gödel proved two important theorems that illustrated the limitations of formal systems. A formal system $\mathcal{L}$ is called "consistent" if you can never prove both a proposition P and its negation $\neg P$ within the system. Gödel showed that "*Any sufficiently strong formal system of arithmetic is incomplete if it is consistent.*" In other words there are sentences P and $\neg P$ such that neither P nor $\neg P$ is provable using the rules of the formal system $\mathcal{L}$. As P and $\neg P$ express contradictory sentences, one of them must be true. So there must be true statements of the formal system $\mathcal{L}$ that can never be proved. Hence Gödel showed that truth and theoremhood (or provability) are distinct concepts.

In a second theorem, Gödel showed that the simple consistency of $\mathcal{L}$ cannot be proved in $\mathcal{L}$. Thus a formal system might be harboring deep-seated contradictions.

The results of Turing and Gödel are startling. They reveal that our commonsense intuitions regarding logical and mathematical theorem proving are not reliable. They are no less startling than the phenomena of entanglement, non-locality, etc in quantum physics.

In the 1980s some scientists began to think about the possible connections between physics and computability [320]. To do so, we must distinguish between Nature, which does what it does, and physics, which provides models of Nature expressed in mathematical form. The fact that physics is a mathematical science means that it is ultimately a formal system. Asher Peres and Wojciech Zurek have articulated three reasonable desiderata of a physical theory [390], namely, determinism, verifiability, and universality (i.e., the theory can describe anything). They conclude that:

"*Although quantum theory is universal, it is not closed. Anything can be described by it, but something must remain unanalyzed. This may not be a flaw of quantum theory: It is likely to emerge as a logical necessity in any theory which is self-referential, as it attempts to describe its own means of verification*."

"*In this sense it is analogous to Gödel's undecidability theorem of formal number theory: the consistency of the system of axioms cannot be verified because there are mathematical statements which can neither be proved nor disproved by the use of the formal rules of the theory, although their truth may be verified by metamathematical reasoning*."

In a later paper Peres points out a "logico-physical paradox" [385]. He shows that it is possible to set up three quantum observables such that two of the observables have to obey the Heisenberg Uncertainty Principle. This Principle, says that certain pairs of observables, such as the position and momentum of a particle, cannot be measured simultaneously. Measuring one such observable necessarily disturbs the complementary observable, so you can never measure both observable together. Nevertheless, Peres arranges things so that he can use the rules of quantum mechanics to predict, with certainty, the value of both these variables individually. Hence we arrive at an example system that we can say things about but which we can never determine experimentally (a physical analogue of Gödel's undecidability theorem).

4.3.4 Proving Versus Providing Proof

Many decades have now passed since Turing first dreamt of his machine and in fact today there are a number of programs around that actually perform as artificial mathematicians in exactly the sense Turing anticipated. Current interest in them stems not only from a wish to build machines that can perform mathematical reasoning but also more general kinds of logical inference such as medical diagnosis, dialog management, and even legal reasoning. Typically, these programs consist of three distinct components: a reservoir of knowledge about some topic (in the form

of axioms and rules of inference), an inference engine (which provides instructions on how to pick which rule to apply next), and a specific conjecture to be proved.

In one of the earliest examples, SHRDLU, a one-armed robot, was given a command in English which was converted into its logical equivalent and then used to create a program to orchestrate the motion of the robot arm [542]. So the robot gave the appearance of understanding a command in plain English simply by following rules for manipulating symbols. Nowadays such capabilities are commonplace. For example, many cell phones can understand a limited repertoire of verbal commands to dial telephone numbers, and some companies use automated query-answering systems to field routine customer enquiries.

In a more sophisticated example, the British Nationality Act was encoded in first-order logic and a theorem prover used to uncover logical inconsistencies in the legislation [447]. Similarly, the form of certain legal arguments can be represented in logic which can then be used to find precedents by revealing analogies between the current case and past examples. So although most people would think themselves far removed from the issue of "theorem proving," they could be in for a surprise if the tax authorities decided to play these games with the tax laws!

Today's artificial mathematicians are far less ingenious than their human counterparts. On the other hand, they are infinitely more patient and diligent. These qualities can sometimes allow artificial mathematicians to churn through proofs upon which no human would have dared embark. Take, for example, the case of map coloring. Cartographers conjectured that they could color any planar map with just four different colors so that no two adjacent regions had the same color. However, this conjecture resisted all attempts to construct a proof for many years. In 1976 the problem was finally solved with the help of an artificial mathematician. The "proof," however, was somewhat unusual in that it ran to some 200 pages [541]. For a human to even check it, let alone generate it, would be a mammoth undertaking. Table 4.4 shows a summary of some notable milestones in mathematical proof by humans and machines.

Despite differences in the "naturalness" of the proofs they find, artificial mathematicians are nevertheless similar to real mathematicians in one important respect: their output is an explicit sequence of reasoning steps (i.e., a proof) that, if followed meticulously, would convince a skeptic that the information in the premises combined with the rules of logical inference would be sufficient to deduce the conclusion. Once such a chain were found the theorem would have been proved. The important point is that the proof chain is a tangible object that can be inspected at leisure. Surprisingly, this is not necessarily the case with a QTM. In principle, a QTM could be used to create some proof that relied upon quantum mechanical interference among all the computations going on in superposition. Upon interrogating the QTM for an answer you might be told, "Your conjecture is true," but there would be no way to exhibit all the computations that had gone on in order to arrive at the conclusion. Thus, for a QTM, the ability to prove something and the ability to provide the proof trace are quite distinct concepts. Worse still, if you tried to peek inside the QTM as it was working, to glean some information about the state of the proof at that time, you would invariably disrupt the future course of the proof.

Table 4.4 Some impressive mathematical proofs created by humans and machines. In some cases simple proofs of long-standing mathematical conjectures have only recently been discovered. In other cases, the shortest known proofs are extremely long, and arguably too complex to be grasped by any single human

Mathematician	Proof feat	Notable features
Daniel Gorenstein	Classification of finite simple groups	Created by human. 15,000 pages long
Kenneth Appel and Wolfgang Haken	Proved the Four Color Theorem	Created by computer. Reduced all planar maps to combinations of 1,936 special cases and then exhaustively checked each case using ad hoc programs. Human mathematicians dislike this proof on the grounds that these ad hoc checking programs may contains bugs and the proof is too hard to verify by hand
Andrew Wiles	Proved Fermat's Last Theorem	Created by human. 200 pages long. Only 0.1% of all mathematicians are competent to judge its veracity
Laszlo Babai and colleagues	Invented probabilistic proof checking	Able to verify that a complex proof is "probably correct" by replicating any error in the proof in many places in the proof, thereby amplifying the chances of the error being detected
Thomas Hales	Proved Kepler's conjecture on the densest way to pack spheres again using ad hoc programs to check a large number of test cases	In reaction to complaints by mathematicians, this proof is now being re-done using automated theorem provers instead of ad hoc checking programs since automated theorem provers, which have been tested extensively, have a higher assurance of being correct
Manindra Agrawal, Neeraj Kayal, and Nitin Saxena	On August 6, 2002 they proved primality testing can done deterministically in polynomial time	Created by humans. Took centuries to find this proof

4.4 Complexity

Complexity theory is concerned with how the inherent cost required to solve a computational problem scales up as larger instances of the problem are considered. It is possible to define many different resources by which the difficulty of performing a computation can be assessed. These include the time needed to perform the computation, the number of elementary steps, the amount of memory used, the number of calls to an oracle or black-box function, and the number of communicative acts. These lead to the notions of computational, query, and communication complexity. Specifically,

- *Computational complexity* measures the number of steps (which is proportional to time) or the minimum amount of memory required (which is proportional to space) needed to solve the problem.
- *Query complexity* measures the number of times a certain sub-routine must be called, or "queried", in order to solve the problem.
- *Communication complexity* measures the volume of data that must be sent back and forth between parties collaborating to solve the problem.

Thus, whereas computability is concerned with which computational tasks computers can and cannot do, *complexity* is concerned with the efficiency with which they can do them. Efficiency is an important consideration for real-world computing. The fact that a computer can solve a particular kind of problem, in principle, does not guarantee that it can solve it in practice. If the running time of the computer is too long, or the memory requirements too great, then an apparently feasible computation can still lay beyond the reach of any practicable computer.

Computer scientists have developed a taxonomy for describing the complexity of various algorithms running on different kinds of computers. The most common measures of efficiency employ the rate of growth of the time or memory needed to solve a problem as the size of the problem increases. Of course "size" is an ambiguous term. Loosely speaking, the "size" of a problem is taken to be the number of bits needed to state the problem to the computer. For example, if an algorithm is being used to factor a large integer N, the "size" of the integer being factored would be roughly $\log_2 N$.

The traditional computational complexity distinction between tractable and intractable problems depends on whether the asymptotic scaling of the algorithm grows polynomially, i.e., $\mathcal{O}(n^k)$, or exponentially, i.e., $\mathcal{O}(k^n)$ with the problem size n.

These notions or tractability and intractability are somewhat imperfect because asymptotic scaling results are unattainable mathematical ideals in a finite Universe. Nor do they take into account the practically interesting range of sizes of problem instances. For example, airline scheduling is an **NP-Complete** problem. In the worst case, the time needed to find the optimal schedule scales exponentially in the number of aircraft to be scheduled. But the number of jetliners with which we are ever likely to have to deal, in practice, is bounded. So if someone invented a scheduling algorithm that scaled as $O(n^{100})$ (where n is the number of jetliners) then, even

though it is polynomial it might not be practically better than an exponential time scheduling algorithm for realistic problems.

The reason complexity classifications are based on the rates of growth of running times and memory requirements, rather than absolute running times and memory requirements, is to factor out the variations in performance experienced by different makes of computers with different amounts of RAM, swap space, and processor speeds. Using a growth rate-based classification, the complexity of a particular algorithm becomes an intrinsic measure of the difficulty of the *problem* the algorithm addresses.

Although complexity measures are independent of the precise make and configuration of computer, they are related to a particular mathematical model of the computer such as a deterministic Turing machine or a probabilistic Turing machine. It is now known, for example, that many problems that are intractable with respect to a deterministic Turing machine can be solved efficiently, or at least can sometimes have their solutions approximated efficiently, with high probability on a probabilistic Turing machine. The Euclidean Traveling Salesman Problem (Euclidean-TSP), e.g., consists of finding a path having minimum Euclidean distance between a set of points in a plane such that the path visits each point exactly once before returning to its starting point. Euclidean-TSP is known to be **NP-Complete** [377], and therefore rapidly becomes intractable as the number of points to be visited, $N \to \infty$. Nevertheless, in [20], Arora exhibits a randomized algorithm that can find a tour to within a factor of $\mathcal{O}(1 + 1/c)$ of the optimal tour (for any constant c) in a time that scales only as $\mathcal{O}(N(\log(N))^{\mathcal{O}(c)})$. This is worse than linear scaling but much better than exponential scaling. Other examples include random walk algorithms for approximating the permanent of a matrix with non-zero entries [255], finding satisfying assignments to a Boolean expression (k-SAT with $k > 2$) [439], estimating the volume of a convex body [162], and estimating graph connectivity [364]. Classical random walks also underpin many standard methods in computational physics, such as Monte Carlo simulations. Thus, randomization can be a powerful tool for rendering intractable problems tractable provided we are content with finding a good approximation to a global optimum or exact solution.

There are many criteria by which you could assess how efficiently a given algorithm solves a given type of problem. For the better part of the century, computer scientists focused on worst-case complexity analyses. These have the advantage that, if you can find an efficient algorithm for solving some problem, in the *worst* case, then you can be sure that you have an efficient algorithm for any instance of such a type of problem.

Worst case analyses can be somewhat misleading however. Recently some computer scientists have developed average case complexity analyses. Moreover, it is possible to understand the finer grain structure of complexity classes and locate regions of especially hard and especially easy problems within a supposedly "hard" class [101, 537, 539]. Nevertheless, one of the key questions is whether some algorithm runs in polynomial time or exponential time.

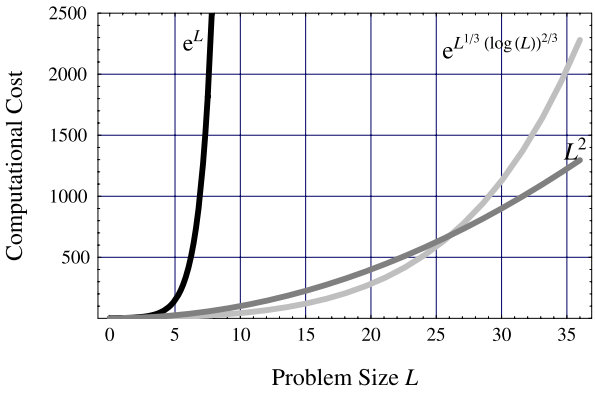

Fig. 4.4 A comparison of polynomial versus exponential growth rates. Exponential growth will always exceed polynomial growth eventually, regardless of the order of the polynomial

4.4.1 Polynomial Versus Exponential Growth

Computer scientists have developed a rigorous way of quantifying the difficulty of a given type of problem. The classification is based on the mathematical form of the function that describes how the computational cost incurred in solving the problem scales up as larger problems are considered. The most important quantitative distinction is between polynomially growing costs (which are deemed tractable) and exponentially growing costs (which are deemed intractable). Exponential growth will always exceed polynomial growth eventually, regardless of the order of the polynomial. For example, Fig. 4.4 compares the growth of the exponential function $\exp(L)$ with the growth of the polynomials L^2, L^3 and L^4. As you can see, eventually, whatever the degree of the polynomial in L, the exponential becomes larger.

A good pair of example problems that illustrate the radical difference between polynomial and exponential growth are multiplication versus factoring. It is relatively easy to multiply two large numbers together to obtain their product, but it is extremely difficult to do the opposite; namely, to find the factors of a composite number:

$$1459 \times 83873 \rightarrow 122370707 \text{ (easy)} \tag{4.1}$$

$$122370707 \rightarrow 1459 \times 83873 \text{ (hard)} \tag{4.2}$$

If, in binary notation, the numbers being multiplied have L bits, then multiplication can be done in a time proportional to L^2, a polynomial in L.

For factoring, the best known classical algorithms are the Multiple Polynomial Quadratic Sieve [460] for numbers involving roughly 100 to 150 decimal digits, and the Number Field Sieve [309] for numbers involving more than roughly 110 decimal digits. The running time of these algorithms grows subexponentially (but superpolynomially) in L, the number of bits needed to specify the number to be factored N. The best factoring algorithms require a time of the order $\mathcal{O}(\exp(L^{\frac{1}{3}}(\log L)^{2/3}))$ which grows subexponentially (but superpolynomially) in L, the number of bits needed to specify the number being factored.

Table 4.5 Progress in factoring large composite integers. One MIP-Year is the computational effort of a machine running at one million instructions per second for one year

Number	Number of decimal digits	First factored	MIPS years
Typical	20	1964	0.001
Typical	45	1974	0.01
Typical	71	1984	0.1
RSA-100	100	1991	
RSA-110	110	1992	
RSA-120	120	1993	825
RSA-129	129	1994	5000
RSA-130	130	1996	750
RSA-140	140	1999	2000
RSA-150	150	2004	
RSA-155	155	1999	8000

Richard Crandall charted the progress in factoring feats from the 1970s to the 1990s [118]. In Table 4.5 we extend his data to more modern times. In the early 1960s computers and algorithms were only good enough to factor numbers with 20 decimal digits, but by 1999 that number had risen to a 155 decimal digit numbers, but only after a Herculean effort. Many of the numbers used in these tests were issued as grand challenge factoring problems by RSA Data Securities, Inc., and hence bear their name. Curiously, RSA-155 was factored prior to RSA-150 (a smaller number). The most famous of these factoring challenge problems is RSA-129.

As we show later in the book, the presumed difficulty of factoring large integers is the basis for the security of so-called public key cryptosystems that are in widespread use today. When one of these systems was invented the authors laid down a challenge prize for anyone who could factor the following 129 digit number (called RSA-129) :

$$\begin{aligned}\text{RSA-129} = 1143816257578888676692357799761466120102182 \\ \ldots 967212423625625618429357069352457338978305 9 \\ \ldots 712356395870505898907514759929002687954354 1\end{aligned} \tag{4.3}$$

But in 1994 a team of computer scientists using a network of workstations succeeding in factoring RSA-129 $= p \times q$ where the factors p are q are given by:

$$\begin{aligned}p = 3490529510847650949147849619903898133417764638493387 8 \\ \ldots 43990820577 \\ q = 3276913299326670954996198819083446141317764296799294 2 \\ \ldots 539798288533\end{aligned} \tag{4.4}$$

Extrapolating the observed trend in factoring suggests that it would take millions of MIP-Years to factor a 200-digit number using conventional computer hardware.

However, it might be possible to do much better than this using special purposes factoring engines as we discuss in Chap. 13.

Although, the traditional computational complexity distinction between tractable and intractable problems depends on whether the asymptotic scaling of the algorithm grows polynomially, i.e., $O(n^k)$, or exponentially, i.e., $O(k^n)$ with the problem size n, strictly speaking, this distinction is imperfect since it does not take into account the finiteness of the Universe. Asymptotic results are unattainable mathematical ideals in a finite Universe. Nor do they take into account the practically interesting range of sizes of problem instances. For example, airline scheduling is an **NP-Complete** problem. In the worst case, the time needed to find the optimal schedule scales exponentially in the number of aircraft to be scheduled. But the number of jetliners with which we are ever likely to have to deal, in practice, is bounded. So if someone invented a scheduling algorithm that scaled as $O(n^{100})$ (where n is the number of jetliners) then, even though it is polynomial it might not be practically better than an exponential time scheduling algorithm for realistic problems.

4.4.2 Big $\mathcal{O}$, Θ and Ω Notation

Complexity theory involves making precise statements about the scaling behavior of algorithms in the asymptotic limit. This is usually described by comparing the growth rate of the algorithm to that of a simple mathematical function in the limit that the size of the computational problem goes to infinity. The most common asymptotic scaling relationships, together with their standard notations, are summarized in Table 4.6.

For example, consider the three functions $f(x) = \sqrt{\frac{x}{2}}$, $g(x) = \frac{3}{x}\sin x + \log x$, and $h(x) = \log \frac{3x}{4}$. Their graphs are shown in Fig. 4.5. For small values of x, $g(x)$ can be greater than or less than $f(x)$, and likewise greater than or less than $h(x)$. However, asymptotically, i.e., "eventually", $g(x)$ is bounded above by $f(x)$ and therefore $g(x) = \mathcal{O}(f(x))$. Similarly, asymptotically, $g(x)$ is bounded below by $h(x)$ and so $g(x) = \Omega(h(x))$. However, as the limit $\lim_{x\to\infty} |\frac{\frac{3}{x}\sin x + \log x}{\log \frac{3x}{4}} - 1| = 0$, we also have $g(x)$ equals $h(x)$ asymptotically, i.e., $g(x) \sim h(x)$ asymptotically.

We can use the aforementioned notation to characterize the asymptotic behaviors of some well-known algorithms. Table 4.7 shows the asymptotic running times of some famous algorithms.

4.4.3 Classical Complexity Zoo

Knowing the exact functional forms for the rates of growth of the number of computational steps for various algorithms allows computer scientists to classify computational problems based on difficulty. The most useful distinctions are based on

Table 4.6 Notation used to characterize the asymptotic scaling behavior of algorithms

Notation	Meaning	Formal definition
$f(x) = \mathcal{O}(g(x))$	$f(x)$ is bounded above by $g(x)$ asymptotically	As $x \to \infty, \exists k$ s.t. $\lvert f(x)\rvert \leq kg(x)$
$f(x) = o(g(x))$	$f(x)$ is dominated by $g(x)$ asymptotically	As $x \to \infty, \forall k$ s.t. $\lvert f(x)\rvert \leq kg(x)$
$f(x) = \Omega(g(x))$	$f(x)$ is bounded below by $g(x)$ asymptotically	As $x \to \infty, \exists k$ s.t. $\lvert f(x)\rvert \geq kg(x)$
$f(x) = \omega(g(x))$	$f(x)$ dominates $g(x)$ asymptotically	As $x \to \infty, \forall k$ s.t. $\lvert f(x)\rvert \geq kg(x)$
$f(x) = \Theta(g(x))$	$f(x)$ is bounded above and below by $g(x)$ asymptotically	As $x \to \infty, \exists k_1, k_2$ s.t. $k_1 g(x) \leq \lvert f(x)\rvert \geq k_2 g(x)$
$f(x) \sim g(x)$	$f(x)$ equals $g(x)$ asymptotically	As $n \to \infty, \forall k$ s.t. $\lvert f(x)/g(x) - 1\rvert \leq k$

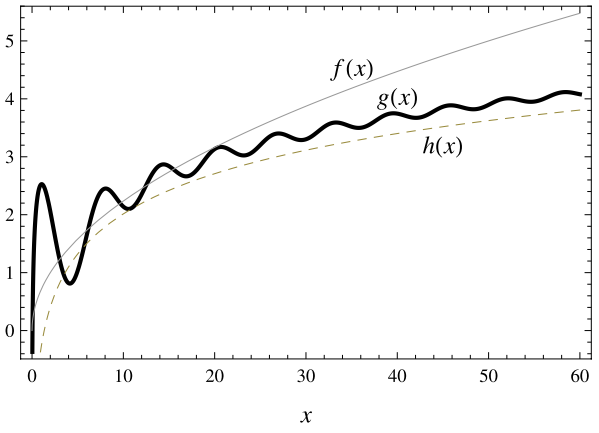

Fig. 4.5 Graphs of $f(x) = \sqrt{\frac{x}{2}}$, $g(x) = \frac{3}{x}\sin x + \log x$, and $h(x) = \log \frac{3x}{4}$. As x becomes larger the relative dominance of the functions becomes clear

classes of problems that either can or cannot be solved in polynomial time, in the worst case. Problems that can be solved in polynomial time are usually deemed "tractable" and are lumped together into the class P. Problems that cannot be solved in polynomial time are usually deemed "intractable" and may be in one of several classes. Of course it is possible that the order of the polynomial is large making a supposedly "tractable" problem rather difficult in practice. Fortunately, such large polynomial growth rates do not arise that often, and the polynomial/exponential distinction is a pretty good indicator of difficulty. In Table 4.8 we list some classical complexity classes.

The known inclusion relationships between the more important of these complexity classes are shown in Fig. 4.6.

Table 4.7 The asymptotic scaling behavior of some important algorithms

Algorithm	Description	Classical	Quantum	Source
UNSTRUCTURED SEARCH	Given a black box function $f(x)$ that returns 1 iff $x = t$ and 0 otherwise how many calls to $f(x)$ are needed to find the index t?	$\Omega(N)$	$\mathcal{O}(\sqrt{N})$	See [219]
FACTORING INTEGERS	Given an integer N find factors p and q such that $N = pq$	With $n = \log N$, $\mathcal{O}(e^{n^{1/3}}(\log n)^{2/3})$	$\mathcal{O}((\log N)^3)$	See [458]
VERIFYING MATRIX PRODUCT	Given matrices A, B, and C, verify $A \cdot B = C$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{5/3})$	See [87]
MINIMUM SPANNING TREE OF WEIGHTED GRAPH	Given an oracle that has knowledge of the adjacency matrix of a graph, G, how many calls to the oracle are required to find a minimum spanning tree?	$\Omega(n^2)$	$\Theta(n^{2/3})$	See [161]
DECIDING GRAPH CONNECTIVITY	Given an oracle that has knowledge of the adjacency matrix of a graph, G, how many calls to the oracle are required to decide if the graph is connected?	$\Omega(n^2)$	$\Theta(n^{2/3})$	See [161]
FINDING LOWEST WEIGHT PATHS	Given an oracle that has knowledge of the adjacency matrix of a graph, G, how many calls to the oracle are required to find a lowest weight path?	$\Omega(n^2)$	$\mathcal{O}(n^{2/3}(\log n)^2)$	See [161]
DECIDING BIPARTITENESS	Given an oracle that has knowledge of the adjacency matrix of a graph, G, how many calls to the oracle are required to decide if the graph is bipartite?	$\Omega(n^2)$	$\mathcal{O}(n^{3/2})$	See [59]

Table 4.8 Some classical complexity classes and example problems within those classes

Classical complexity class	Intuitive meaning	Examples
P or **PTIME**	**P**olynomial-Time: the running time of the algorithm is, in the worst case, a polynomial in the size of the input. All problems in P are tractable	Multiplication, linear programming [276], and primality testing (a relatively new addition to this class) [5, 6]. Computing the determinant of a matrix. Deciding if a graph *has* a perfect matching
ZPP	**Z**ero-Error **P**robabilistic **P**olynomial-Time: Can be solved, with certainty, by PTMs in average case polynomial time	Randomized Quicksort
BPP	**B**ounded-Error **P**robabilistic **P**olynomial Time: Decisions problems solvable in polynomial time by PTMs with probability $> 2/3$. Probability of success can be made arbitrarily close to 1 by iterating the algorithm a certain number of times	Decision version of Min-Cut [198]
NP	**N**ondeterministic **P**olynomial time: The class of decision problems with the property that if you could magically "guess" a correct solution you could verify this fact in polynomial time	Factoring composite integers: a purported solution can be verified by multiplying the claimed factors and comparing the result to the number being factored. At the present time it is unknown whether or not $\mathbf{P} = \mathbf{NP}$ but it appears unlikely
NP-Complete	Subset of problems in **NP** that can be mapped into one another in polynomial time. If just one of the problems in this class is shown to be tractable, then they must all be tractable. Not all problems in **NP** are **NP-Complete**	Examples include Scheduling, Satisfiability, Traveling Salesman Problem, 3-Coloring, Subset-Sum, Hamiltonian Cycle, Maximum Clique [115]
NP-Hard	The optimization version of **NP-Complete** problems, wherein one not only wants to decide if a solution exists but to actually one	Determining the solutions to a SAT problem
# P	Counting version of an **NP-Hard** problem	Determining the number of satisfying assignments to a SAT problem [507]
# P-Complete	**Sharp P Complete**	Computing the permanent of an $n \times n$ 0-1 matrix $\{a_{ij}\}$, i.e., $\sum_{\sigma} \prod_{i=1}^{n} a_{i,\sigma(i)}$ where σ ranges over all permutations of $1, 2, 3, \ldots, n$. The number of perfect matchings in a graph

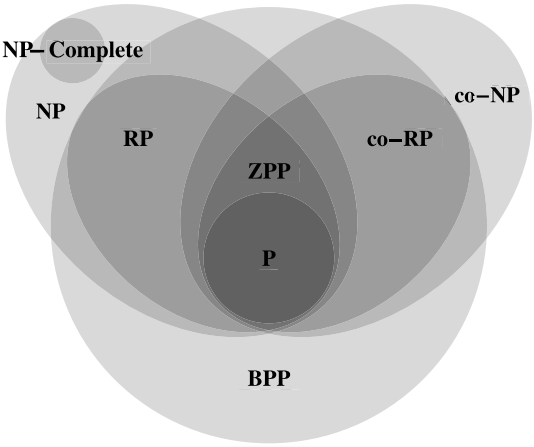

Fig. 4.6 Some known inclusion relationships between classical complexity classes. The most important classes shown are **P**—class of problems that can be solved in polynomial time, and **NP**—the class of problems whose solution can be verified in polynomial time. Of these a special subset—the **NP-Complete** problems—are at least as hard as any other problem in **NP**

4.4.4 Quantum Complexity Zoo

The introduction of quantum considerations turns out to have profound implications for the foundations of computer science and information theory. Decades of old theory must now be taken from the library shelves, dusted off and checked for an implicit reliance upon classical bits and classical physics. By exploiting entirely new kinds of physical phenomena, such as superposition, interference, entanglement, non-determinism and non-clonability, we can suddenly catch a glimpse of a new theoretical landscape before us. This shift from classical to quantum is a qualitative change not merely a quantitative change such as the trends we saw in Chap. 1. It is something entirely new.

Just as there are classical complexity classes, so too are there quantum complexity classes (see Fig. 4.7). As quantum Turing machines are quantum mechanical generalizations of probabilistic Turing machines, the quantum complexity classes resemble the probabilistic complexity classes. There is a tradeoff between the certainty of your answer being correct versus the certainty of the answer being available within a certain time bound. In particular, the classical classes **P**, **ZPP**, and **BPP** become the quantum classes **QP**, **ZQP**, and **BQP**. These mean, respectively, that a problem can be solved with certainty in *worst*-case polynomial time, with certainty in *average*-case polynomial time, and with *probability greater than* $2/3$ in worst-case polynomial time, by a quantum Turing machine.

Statements about the relative power of one type of computer over another can be couched in the form of subset relationships among complexity classes. Thus **QP** is the class of problems that can be solved, with certainty, in polynomial time, on a quantum computer, and **P** is the set of problems that can be solved, with certainty, in polynomial time on a classical computer. As the class **QP** contains the class **P** (see Table 4.9) this means that there are more problems that can be solved efficiently by a quantum computer than by any classical computer. Similar relationships are now known for some of the other complexity classes too, but there are still many open questions remaining.

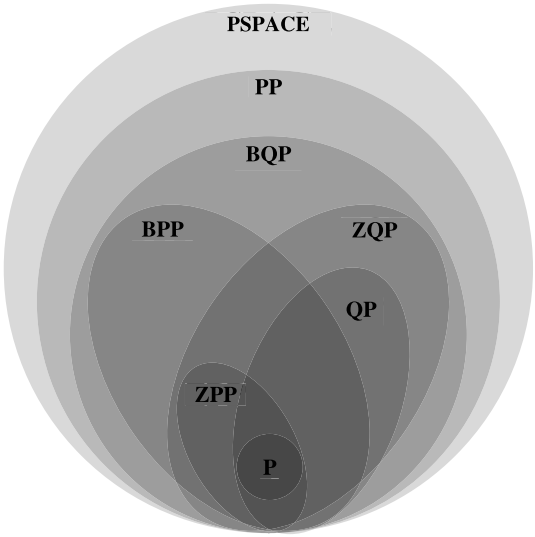

Fig. 4.7 Some known inclusion relationships between classical and quantum complexity classes. Classes correspond to circular and oval shapes and containment is shown by shape inclusion. The most important classes shown are **QP**—the class of problems that can be solved with certainty by a quantum computer in *worst*-case polynomial time; **ZQP**—the class of problems that can be solved with certainty by a quantum computer in *average*-case polynomial time; and **BQP**—the class of problems that can be solved with *probability greater than* $2/3$ by a quantum computer in worst-case polynomial time

The study of quantum complexity classes began with David Deutsch in his original paper on quantum Turing machines (QTMs). The development of the field is summarized in Table 4.10.

In Deutsch's original paper he presented the idea of quantum parallelism. Quantum parallelism allows you to compute an exponential number of function evaluations in the time it takes to do just one function evaluation classically. Unfortunately, the laws of quantum mechanics make it impossible to extract more than one of these answers explicitly. The problem is that although you can indeed calculate all the function values for all possible inputs at once, when you read off the final answer from the tape, you will only obtain one of the many outputs. Worse still, in the process, the information about all the other outputs is lost irretrievably. So the net effect is that you are no better off than had you used a classical Turing machine. So, as far as function evaluation goes, the quantum computer is no better than a classical computer.

Deutsch realized that you could calculate certain joint properties of all of the answers without having to reveal any one answer explicitly. (We explained how this works in Chap. 1). The example Deutsch gave concerned computing the XOR (exclusive-or) of two outputs. Suppose there is a function f that can receive one of two inputs, 0 or 1, and that we are interested in computing the XOR of both function values, i.e., $f(0) \oplus f(1)$ (where $\oplus$ here means "XOR"). The result could,

Table 4.9 Some quantum complexity classes and their relationships to classical complexity classes

Quantum class	Class of computational problems that can…	Relationship to classical complexity classes (if known)
QP	**Q**uantum **P**olynomial-Time: … be solved, with certainty, in worst-case polynomial time by a quantum computer. All problems in **QP** are tractable	$\mathbf{P} \subset \mathbf{QP}$ (The quantum computer can solve more problems in worst case polynomial time than the classical computer)
ZQP	**Z**ero-Error **Q**uantum **P**olynomial-Time: … can be solved, with zero error probability, in expected polynomial time by a quantum computer	$\mathbf{ZPP} \subset \mathbf{ZQP}$
BQP	**B**ounded-Error **Q**uantum **P**olynomial Time: … be solved in worst-case polynomial time by a quantum computer with probability $> \frac{2}{3}$ (thus the probability of error is bounded; hence the **B** in **BQP**)	$\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{PSPACE}$ (i.e., the possibility of the equality means it is not known whether QTMs are more powerful than PTMs.) **BQP** is the class of problems that are easy for a quantum computer, e.g., factoring composite integers, computing discrete logarithms, sampling from a Fourier transform, estimating eigenvalues, and solving Pell's equation [225, 458]

for example, be a decision as to whether to make some stock investment tomorrow based on today's closing prices. Now suppose that, classically, it takes 24 hours to evaluate each f. Thus if we are stuck with a single classical computer, we would never be able to compute the XOR operation in time to make the investment the next day. On the other hand, using quantum parallelism, Deutsch showed that half the time we would get no answer at all, and half the time we would get the guaranteed correct value of $f(0) \oplus f(1)$. Thus the quantum computer would give useful advice half the time and never give wrong advice.

Richard Jozsa refined Deutsch's ideas about quantum parallelism by showing that many functions—for example, SAT (the propositional satisfiability problem)—cannot be computed by quantum parallelism at all [261]. Nevertheless, the question about the utility of quantum parallelism for tackling computational tasks that were not function calculations remained open.

In 1992 Deutsch and Jozsa exhibited a problem, that was not equivalent to a function evaluation, for which a quantum Turning machine (QTM) was exponentially faster than a classical deterministic Turing Machine (DTM). The problem was rather contrived, and consisted of finding a true statement in a list of two statements. It was possible that both statements were true, in which case either statement would be acceptable as the answer. This potential multiplicity of solutions meant that the problem could not be reformulated as a function evaluation. The upshot was that the QTM could solve the problem in a "polynomial in the logarithm of the prob-

Table 4.10 Historical development of quantum complexity theory

Year	Advance in quantum complexity theory
Benioff (1980)	Shows how to use quantum mechanics to implement a Turing Machine (TM)
Feynman (1982)	Shows that TMs cannot simulate quantum mechanics without exponential slowdown
Deutsch (1985)	Proposes first universal QTM and the method of quantum parallelism. Proves that QTMs are in the same complexity with respect to function evaluation as TMs. Remarks that some computational tasks (e.g, random number generation) do not require function evaluation, Exhibits a contrived decision problem that can be solved faster on a QTM than on a TM
Jozsa (1991)	Describes classes of functions that can and cannot be computed efficiently by quantum parallelism
Deutsch & Jozsa (1992)	Exhibit a contrived problem that the QTM solves with certainty in poly-log time, but that requires linear time on a DTM. Thus, the QTM is exponentially faster than the DTM. Unfortunately, the problem is also easy for a PTM so this is not a complete victory over classical machines
Berthiaume & Brassard (1992)	Prove $P \subset QP$ (strict inclusion). The first definitive complexity separation between classical and quantum computers
Bernstein & Vazirani (1993)	Describe a universal QTM that can simulate any other QTM efficiently (Deutsch's QTM could simulate other QTMs, but only with an exponential slowdown)
Yao (1993)	Shows that complexity theory for quantum circuits matches that of QTMs. This legitimizes the study of quantum circuits (which are simpler to design and analyze than QTMs)
Berthiaume & Brassard (1994)	Prove that randomness alone is not what gives QTMs the edge over TMs. Prove that there is a decision problem that is solved in polynomial time by a QTM, but requires exponential time, in the worst case, on a DTM and PTM. First time anyone showed a QTM to beat a PTM. Prove there is a decision problem that is solved in exponential time on a QTM but which requires double exponential times on a DTM on all but a few instances
Simon (1994)	Lays foundational work for Shor's algorithm
Shor (1994)	Discovers a polynomial-time quantum algorithm for factoring large composite integers. This is the first *significant* problem for which a quantum computer is shown to outperform any type of classical computer. Factoring is related to breaking codes in widespread use today
Grover (1996)	Discovers a quantum algorithm for finding a single item in an unsorted database in square root of the time it would take on a classical computer. if the search takes N steps classically, it takes $(\pi/4)\sqrt{N}$ quantum-mechanically

lem size" time (poly-log time), but that the DTM required linear time. Thus the QTM was exponentially faster than the DTM. The result was only a partial success,

however, as a probabilistic Turing machine (PTM) could solve it as efficiently as could the QTM. But this did show that a quantum computer at least could beat a deterministic classical computer.

So now the race was on to find a problem for which the QTM beat a DTM and a PTM. Ethan Bernstein and Umesh Vazirani analyzed the computational power of a QTM and found a problem that did beat both a DTM and a PTM [57]. Given any Boolean function on n-bits Bernstein and Vazirani showed how to sample from the Fourier spectrum of the function in polynomial time on a QTM. It was not known if this were possible on a PTM. This was the first result that hinted that QTMs might be more powerful than PTMs.

The superiority of the QTM was finally clinched by André Berthiaume and Gilles Brassard who constructed an "oracle" relative to which there was a decision problem that could be solved with certainty in worst-case polynomial time on the quantum computer, yet cannot be solved classically in probabilistic expected polynomial time (if errors are not tolerated). Moreover, they also showed that there is a decision problem that can be solved in exponential time on the quantum computer, that requires double exponential time on all but finitely many instances on any classical deterministic computer. This result was proof that a quantum computer could beat both a deterministic and probabilistic classical computer but it was still not headline news because the problems for which the quantum computer was better were all rather contrived.

The situation changed when, in 1994, Peter Shor, building on work by Dan Simon, devised polynomial-time algorithms for factoring composite integers and computing discrete logarithms. The latter two problems are believed to be intractable for any classical computer, deterministic or probabilistic. But more important, the factoring problem is intimately connected with the ability to break the RSA cryptosystem that is in widespread use today. Thus if a quantum computer could break RSA, then a great deal of sensitive information would suddenly become vulnerable, at least in principle. Whether it is vulnerable in practice depends, of course, on the feasibility of designs for actual quantum computers.

4.5 What Are Possible "Killer-Aps" for Quantum Computers?

The discovery of Shor's and Grover's algorithms led many people to expect other quantum algorithms would quickly be found. However, this was not the case. It turns out to be quite hard to find new quantum algorithms. So where exactly should we be looking? Currently, there are two broad classes of quantum algorithms. There are those, such as Shor's algorithm, that exhibit *exponential* improvements over what is possible classically and those, such as Grover's algorithm, that exhibit *polynomial* speedups over what is possible classically. Shor's algorithm is arguably the more interesting case since exponential speedups are game-changing. It is natural to wonder whether the other computational problems that lie in the same complexity class as the problems tackled by Shor's algorithm might be amenable to a similar speedup.

The most likely candidate opportunities are therefore computational problems (like factoring and discrete log) that are believed to be in the **NP-Intermediate** class. These are problems that are certainly in **NP** but neither in **P** nor **NP-Complete**. Some examples of presumed **NP-Intermediate** problems collected by Miklos Santha are as follows [429]:

GRAPH-ISOMORPHISM Given two graphs $G_1 = (V, E_1)$, and $G_2 = (V, E_2)$, is there a mapping between vertices, $f : V \to V$, such that $\{u, v\} \in E_1 \Leftrightarrow \{f(u), f(v)\} \in E_2$?

HIDDEN-SUBGROUP Let G be a finite group, and let $\gamma : G \to X$ (X a finite set), such that γ is constant and distinct on cosets of a subgroup H of G. Find a generating set for H.

PIGEONHOLE SUBSET-SUM Given a set of positive integers $s_1, s_2, \ldots, s_n \in \mathbb{N}$ such that $\sum_{i=1}^{n} s_i < 2^n$, are there two subsets of indices, $I_1 \neq I_2 \subseteq \{1, 2, \ldots, n\}$ that sum to the same value, i.e., $\sum_{i \in I_1} s_i = \sum_{j \in I_2} s_j$?

With sufficient research, it is conceivable any of the **NP-Intermediate** problems might be re-classified at some point. Nevertheless, today, the **NP-Intermediate** problems are the best prospects for being amenable to an exponential speedup using some as-yet-to-be-discovered quantum algorithm. So far, exponentially faster quantum algorithms have been found for solving the Hidden Subgroup Problem over abelian groups [72, 283, 362, 458] and some non-abelian groups [30, 192]. However, extending these results to other non-abelian groups has proven to be challenging and only limited progress has been made [324]. Researchers are especially interested in extending these results to two families of non-abelian groups—permutation groups and dihedral groups—because doing so will lead immediately to efficient solutions for GRAPH ISOMORPHISM [262] and the SHORTEST LATTICE VECTOR problems [416], which would make quantum computing considerably more interesting.

While progress is therefore being made the exact boundary where quantum algorithms can be found that outperform classical counterparts by an exponential factor is still ill-defined.

4.6 Summary

The most important concept of this chapter is the idea that, as computers are physical objects, their capabilities are constrained exclusively by the laws of *physics* and not pure mathematics. Yet the current (classical) theory of computation had several independent roots, all based on *mathematical* idealizations of the computational process. The fact that these mathematically idealized models turned out to be equivalent to one another led most classical computer scientists to believe that the key elements of computation had been captured correctly, and that it was largely a

matter of taste as to which model of computation to use when assessing the limits of computation.

However, it turns out that the classical models of computation all harbor implicit assumptions about the physical phenomena available to the computer. As Feynman and Deutsch pointed out, models of computation that allow for the exploitation of *quantum* physical effects are qualitatively different from, and potentially more powerful than, those that do not. Which quantum effects really matter the most is still not entirely understood, but the phenomenon of entanglement appears to play a significant role.

In this chapter we surveyed issues of complexity, computability, and universality in the quantum and classical domains. Although there is no *function* a quantum computer can compute that a classical computer cannot also compute, given enough time and memory, there are computational *tasks*, such as generating true random numbers and teleporting information, that quantum computers can do but which classical ones cannot.

A question of some practical importance is to determine the class of computational problems that quantum computers can solve faster than classical ones. To this end, quantum computer scientists have determined the scaling of the "cost" (in terms of space, time, or communications) of certain quantum algorithms (such as factoring integers, and unstructured search, in comparison to that of their best classical counterparts. Some quantum algorithms, such as Shor's algorithm for factoring composite integers and computing discrete logarithms, Hallgren's algorithm for solving Pell's equation, and eigenvalue estimation, show *exponential* speedups, whereas others, such as Grover's algorithm for unstructured search, show only *polynomial* speedups [55]. The greatest challenge to quantum computer scientists is to systematically expand the repertoire of problems exhibiting exponential speedups. Good candidates for problems that might admit such speedups are the other problems in the same complexity class as FACTORING and DISCRETE-LOG, i.e., **NP-Intermediate**. However, to date, no one has succeeded in showing exponential speedups on these other **NP-Intermediate** problems in their most general form. Other problems admit only a polynomial speedup (e.g., SEARCHING-A-SORTED-LIST) or no speedup whatsoever (e.g., PARITY). So far, no quantum algorithm has been found that can speedup the solution of an **NP-Complete** or **NP-Hard** problem by an exponential factor, and most quantum computer scientists are highly skeptical any such algorithm exists.

4.7 Exercises

4.1 Stirling's approximation for the factorial function is $n! = \Theta(\sqrt{2\pi n}(\frac{n}{e})^n)$ (for integer values of n). Does this mean that $n!$ grows at a faster, slower, or equal rate to $\sqrt{2\pi n}(\frac{n}{e})^n$? Plot a graph of the ratio of the left and right hand sides of Stirling's formula for $n = 1, 2, \ldots, 20$. How does the percentage error in the approximation change with increasing values of n?

4.2 Prove, using non-numeric methods,

(a) The base of natural logarithms, e, and π satisfy $e^{\pi} > \pi^{e}$
(b) The golden ratio $\phi = (1 + \sqrt{5})/2$ is less than $\pi^2/6$. [Hint: $\frac{\pi^2}{6} = \sum_{n=1}^{\infty} \frac{1}{n^2}$]

4.3 Classify the following particular claims involving $\mathcal{O}(\cdot)$ notation as correct or incorrect, and if incorrect, give a corrected version:

(a) $\mathcal{O}(n^3 + n^5) = \mathcal{O}(n^3) + \mathcal{O}(n^5)$
(b) $\mathcal{O}(n^2 \times \log n) = \mathcal{O}(n^2) \times \mathcal{O}(\log n)$
(c) $0.0001n^3 + 1000n^{2.99} + 17 = \mathcal{O}(n^3)$
(d) $4n^4 + 3n^{3.2} + 13n^{2.1} = \mathcal{O}(n^{7.2})$
(e) $\log n^{10} = \mathcal{O}(\log n)$
(f) $(\log n)^{10} = \mathcal{O}(n^{2.1})$
(g) $3\log_{10} n^2 + 10\log_2 \log_2 n^10 = \mathcal{O}(\log_e n)$

4.4 Classify the following generic claims regarding $\mathcal{O}(\cdot)$ notation as correct or incorrect, and if incorrect, give a corrected version:

(a) If $f(n) = \mathcal{O}(g(n))$ then $kf(n) = O(g(n))$ for any k
(b) If $f(n) = \mathcal{O}(g(n))$ and $h(n) = \mathcal{O}(g'(n))$ then $f(n) + h(n) = \mathcal{O}(g(n) + g'(n))$
(c) If $f(n) = \mathcal{O}(g(n))$ and $h(n) = \mathcal{O}(g'(n))$ then $f(n)h(n) = \mathcal{O}(g(n)g'(n))$
(d) If $f(n) = \mathcal{O}(g(n))$ and $g(n) = O(h(n))$ then $f(n) = \mathcal{O}(h(n))$
(e) If $f(n)$ is a polynomial of degree d, then $f(n) = O(n^d)$
(f) If $\log n^k = \mathcal{O}(\log n)$ for $k > 0$
(g) If $(\log n)^k = \mathcal{O}(n^j)$ for $k > 0$ and $j > 0$

4.5 What can be said about the expression $3n^4 + 5n^{2.5} + 14\log n^{1.2}$ in terms of

(a) $\mathcal{O}(\cdot)$ notation
(b) $\Theta(\cdot)$ notation
(c) $\Omega(\cdot)$ notation

4.6 Complexity analyses often involve summing series over finitely many terms. Evaluate the following sums in closed form:

(a) $\sum_{i=1}^{n} i^2$
(b) $\sum_{i=1}^{n} i^3$
(c) $\sum_{i=1}^{n} i^k$
(d) $\sum_{i=1}^{n} 2^i$
(e) $\sum_{i=1}^{n} k^i$
(f) $\sum_{i=1}^{n} i^3 e^i$ where $e \approx 2.71828$

4.7 The following question is aimed at stimulating discussion. It is often said that physicists are searching for a unified theory of physics—an ultimate theory that will explain everything that can be explained. Do you think a unified theory of physics will be expressed mathematically? Will it be a computable axiomatic system pow-

erful enough to describe the arithmetic of the natural numbers? If so, in light of Gödel's Incompleteness theorem, do you think a unified theory of physics is possible? Or will certain truths of the theory be forever beyond proof? That is, if the unified theory of physics is consistent must it be incomplete? And can the consistency of the axioms of the unified theory of physics be proven within the theory?