1 Introduction

1.1 What Is Complexity Theory?

Complexity theory – is it a discipline for theoreticians who have no concern for "the real world" or a central topic of modern computer science?

In this introductory text, complexity theory is presented as an active area of computer science with results that have implications for the development and use of algorithms. Our study will lead to insights into the structure of important optimization problems and will explore the borders of what is algorithmically "possible" with reasonable resources. Since this text is also especially directed toward those who do not wish to make complexity theory their specialty, results that do not (yet) have a connection to algorithmic applications will be omitted.

The areas of complexity theory on the one hand and of the design and analysis of efficient algorithms on the other look at algorithmic problems from two opposing perspectives. An efficient algorithm can be directly applied to solve a problem and is itself a proof of the efficient solvability of the problem. In contrast, in complexity theory the goal is to prove that difficult problems cannot be solved with modest resources. Bearers of bad news are seldom welcome, and so it is that the results of complexity theory are more difficult to communicate than a better algorithm for an important problem. Those who do complexity theory are often asked such questions as

- "Why are you pleased with a proof that a problem is algorithmically difficult? It would be better if it had an efficient algorithmic solution."
- "What good are these results? For my particular applied problem I need an algorithmic solution. Now what do I do?"

Naturally, it would be preferable if a problem proved to be efficiently algorithmically solvable. But whether or not this is the case is not up to us. Once we have agreed upon the rules of the game (roughly: computers, but more about that later), every problem has a well-defined algorithmic complexity. Complexity theory and algorithm theory are both striving to estimate this algorithmic complexity and so to "discover the truth". In this sense, the joy over a proof that a problem is not efficiently solvable is, just like the joy over the design of an efficient algorithm, the joy of finding out more about the true algorithmic complexity.

Of course, our reaction to the discovery of truths does depend on whether hopes were fulfilled or fears confirmed. What are the consequences when we find out that the problem we are investigating is not efficiently solvable? First, there is the obvious and very practical consequence that we can with good reason abandon the search for an efficient algorithm. We need no longer waste our time with attempts to obtain an unreachable goal. We are familiar with this from other sciences as well. Reasonable people no longer build "perpetual motion machines", and they no longer try to construct from a circle, using only straight edge and compass, a square with the same area (the proverbial quadrature of the circle). In general, however, people have a hard time with impossibility results. This can be seen in the large number of suggested designs for perpetual motion machines and the large number of attempts to square a circle that are still being made.

Once we have understood that we must accept negative results as well as positive results, and that they save us unnecessary work, we are left with the question of what to do. In the end, we are dealing with an algorithmic problem the solution to which is important for some particular application. Fortunately, problems in most applications are not unalterably determined. It is often tempting to formulate a problem in a very general form and to place very strict demands on the quality of the solution. If such a general formulation has an efficient solution, great. But when this is not the case, we can often specialize the problem (graphs that model street systems will have low degree because there is a limit on the number of streets that can meet at a single intersection), or perhaps a weaker form of solution will suffice (almost optimal may be good enough). In this way we come up with new problems which are perhaps efficiently algorithmically solvable. And so impossibility proofs (negative results) help us find the problems that are (perhaps "just barely") efficiently solvable.

So complexity theory and the design and analysis of efficient algorithms are the two areas of computer science which together fathom the borders between what can and cannot be done algorithmically with realistic resource requirements. There is, of course, a good deal of "cross-pollination" between the two areas. Often attempts to prove the impossibility of an efficient solution to a problem have so illuminated the structure of the problem that efficient algorithms have been the result. On the other hand, failed attempts to design an efficient algorithm often reveal just where the difficulty of a particular problem lies. This can lead to ideas for proving the difficulty of the problem. It is very often the case that one begins with a false conjecture about the degree of difficulty of a problem, so we can expect to encounter startling results in our study of the complexity of problems.

As a result of this introductory discussion we maintain that

The goal of complexity theory is to prove for important problems that their solutions require certain minimum resources. The results of complexity theory have specific implications for the development of algorithms for practical applications.

We have up until now been emphasizing the relationship between the areas of complexity theory and algorithm design. Now, however, we want to take a look at the differences between these areas. When designing an algorithm we "only" need to develop and analyze *one* algorithm. This provides an *upper bound* for the minimal resource requirements with which the problem can be solved. Complexity theory must provide *lower bounds* for the minimally necessary resource requirements that *every* algorithm that solves the problem must use. For the proof of an upper bound, it is sufficient to design and analyze a *single* algorithm (and algorithms are often designed to support the subsequent analysis). Every lower bound, on the other hand, is a statement about *all* algorithms that solve a particular problem. The set of all algorithms for a problem is not a very structured set. Its only structural characteristic is that the problem be solved. How can we make use of this characteristic? An obvious way to start is to derive from the structure of the problem statements that restrict the set of algorithms we must consider. A specific example: It seems clear that the best algorithms for matrix multiplication do not begin by subtracting matrix elements from each other. But how does one prove this? Or is a proof unnecessary, since the claim is so obvious? Quite the opposite: The best algorithms known for matrix multiplication do in fact begin by subtracting matrix elements (see, for example, Schönhage, Grotefeld, and Vetter (1999)). This clearly shows the danger in drawing very "obvious" but false conclusions. Therefore,

In order to prove that the solution of a particular problem requires certain minimal resources, all algorithms for the problem must be considered. This is the source of the main difficulty that impedes achieving the goals of complexity theory.

We now know what kind of results we desire, and we have indicated that they are difficult to come by. It sounds as if we want to excuse in advance the absence of results. This is indeed the case:

None of the most important problems in complexity theory have been solved, but along the way to answering the central questions many notable results have been achieved.

How do we imagine this situation? The cover of the classic book by Hopcroft and Ullman (1979), which includes an introduction to complexity theory, shows a picture in which a curtain in front of the collection of truths of complexity theory is being lifted with the help of various results, thus allowing a clear view of the results. From our perspective of complexity theory, the curtain has so far only been pushed aside a bit at the edges, so that we can clearly see some "smaller truths". Otherwise, the opaque curtain has been replaced by a thinner curtain through which we can recognize a large portion of the truth, but only in outline and with no certainty that we are not falling prey to an optical illusion.

What does that mean concretely? Problems that are viewed as difficult have not actually been proved to be difficult, but it has been shown that thousands of problems are essentially equally difficult (in a sense that will be made precise later). An efficient solution to any one of these thousands of problems implies an efficient solution to all the others. Or stated another way: a proof that any one of these problems is not efficiently solvable implies that none of them is. Thousands of secrets have joined together to form one great mystery, the unmasking of which reveals all the secrets. In this sense, each of these secrets is just as central as every other and just as important as the great mystery, which we will later refer to as the $\mathsf{NP} \neq \mathsf{P}$-problem. In contrast to many other areas of computer science,

Complexity theory has in the $\mathsf{NP} \neq \mathsf{P}$*-problem a central challenge.*

The advantage of such an important and central problem is that along the way to its solution many important results, methods, and even new research areas are discovered. The disadvantage is that the solution of the central problem may be a long time in coming. We can learn something of this from the 350-year search for a proof of Fermat's Last Theorem (Singh (1998) is recommended for more about that topic). Along the way to the solution, deep mathematical theories were developed but also many false paths were followed. Only because of the notoriety of Fermat's Last Theorem was so much effort expended toward the solution to the problem. The $\mathsf{NP} \neq \mathsf{P}$-problem has taken on a similar role in computer science – but with an unfortunate difference: Fermat's Last Theorem (which says that there are no natural numbers x, y, z, and n with $n \geq 3$ such that $x^n + y^n = z^n$) can be understood by most people. It is fascinating that a conjecture that is so simple to formulate occupied the world of mathematics for centuries. For the role of computer science, it would be nice if it were equally simple to explain to a majority of people the complexity class P and especially NP, and the meaning of the $\mathsf{NP} \neq \mathsf{P}$-problem. Alas, this is not the case.

We will see that in the vicinity of the $\mathsf{NP} \neq \mathsf{P}$-problem important and beautiful results have been achieved. But we must also fear that much time may pass before the $\mathsf{NP} \neq \mathsf{P}$-problem is solved. For this reason, it is not necessarily the best strategy to aim directly for a solution to the problem. Yao (2001) compared our starting position to the situation of those who 200 years ago dreamed of reaching the moon. The strategy of climbing the nearest tree or mountain brings us closer to the moon, but it doesn't really bring us any closer to the goal of reaching the moon. The better strategy was to develop ever better means of transportation (bicycles, automobiles, airplanes, rockets). Each of these intermediate steps represented an earth moving discovery. So it is with complexity theory at the beginning of the third millennium: we must search for intermediate steps and follow suitable paths, even though we can never be certain that they will lead to our goal.

Just as those who worked on Fermat's Last Theorem were "sure" that the conjecture was true, so it is that today the experts believe that $\mathsf{NP} \neq \mathsf{P}$ and, therefore, that all of the essentially equally difficult problems mentioned above are not efficiently solvable. Why is this so? From the opposite assumption that $\mathsf{NP} = \mathsf{P}$ one can derive consequences that contradict all our convictions, even though they have not been proven false. Strassen (1996) has gone so far as to elevate the $\mathsf{NP} \neq \mathsf{P}$-conjecture above the status of a mathematical conjecture and compared it with a physical law (such as $E = mc^2$). This, by the way, opens up the possibility that the hypothesis that $\mathsf{NP} \neq \mathsf{P}$ is true but not provable with our proof techniques. But at this point we are far from being able to discuss this background seriously. Our main conclusion is that it is reasonable to build a theory under the hypothesis that $\mathsf{NP} \neq \mathsf{P}$.

> *Many results in complexity theory assume solidly based but unproven hypotheses, such as* $\mathsf{NP} \neq \mathsf{P}$.

But what if $\mathsf{NP} = \mathsf{P}$? Well, then we must make fundamental modifications to many of our intuitions. Many of the results discussed here would in this case have other interpretations, but most would not become worthless. In general, complexity theory forms an intellectual challenge that differs from the demands of other areas of computer science. Complexity theory takes its place in the scientific landscape among those disciplines that

> *seek to probe the boundaries of what is possible with available resources.*

Here the resources are such things as computation time and storage space. Anyone who is interested in the boundaries of what is (and is not) practically feasible with computers will find that complexity theory provides important answers. But those who come to complexity theory only wanting to know pragmatically if the problem they are interested in can be efficiently solved have also come to the right place.

1.2 Didactic Background

The main goal of this text is to provide as many as possible with a comfortable introduction to modern complexity theory. To this end a number of decisions were made with the result that this text differs from other books on the subject.

Since complexity theory is a polished theory with many branches, some selection of topics is unavoidable. In our selection, we have placed a premium on choosing topics that have a concrete relationship to algorithmic problems. After all, we want the importance of complexity theory for modern computer science to be clear. This comes at the cost of structural and abstract branches of complexity theory, which are largely omitted. In Section 1.3 we discuss in more detail just which topics are covered.

We have already discussed the difficulties of dealing with negative results and the relationship to the area of algorithm design. With a consistent perspective that is markedly algorithmic, we will – whenever it is possible and reasonable – present first positive results and only then derive consequences of negative results. For this reason, we will often quantify results which are typically presented only qualitatively.

In the end, it is the concept of nondeterminism that presents a large hurdle that one must clear in order to begin the study of complexity theory. The usual approach is to first describe nondeterministic computers which "guess" the correct computation path, and therefore can not actually be constructed. We have chosen instead to present randomization as the key concept. Randomized algorithms can be realized on normal computers and the modern development of algorithms has clearly shown the advantages of randomized algorithms (see Motwani and Raghavan (1995)). Nondeterminism then becomes a special case of randomization and therefore an algorithmically realizable concept, albeit one with an unacceptable probability of error (see Wegener (2002)). Using this approach it is easy to derive the usual characterizations of nondeterminism later.

We will, of course, give complete and formal proofs of our results, but often there are ugly details that make the proofs long and opaque. The essential ideas, however, are usually shorter to describe and much clearer. So we will include, in addition to the proofs, discussions of the ideas, methods, and concepts involved, in the hope that the interplay of all components will ease the introduction to complexity theory.

1.3 Overview

In Section 1.1 we simplified things by assuming that a problem is either algorithmically difficult or efficiently solvable. All concepts that are not formally defined must be uniquely specified. This begins already with the concept of an algorithmic problem. Doesn't the difficulty of a problem depend on just how one formulates the problem and on the manner in which the necessary data are made available? In Chapter 2 we will clarify essential notions such as algorithmic problem, computer, computation time, and algorithmic complexity. So that we can talk about some example problems, several important algorithmic problems and their variants will also be introduced and motivated. To avoid breaking up the flow of the text, a thorough introduction to O-notation has been relegated to the appendix.

In Chapter 3 we introduce randomization. We discuss why randomized algorithms are for many applications an extremely useful generalization of deterministic algorithms – provided the probability of undesirable results (such as computation times that are too long or an incorrect output) is vanishingly small. The necessary results from probability theory are introduced, proved, and clarified in an appendix. In the end we arrive at the classes of problems that we consider to be efficiently solvable.

The number of practically relevant algorithmic problems is in the thousands, and we would despair if we had to treat each one independently of the others. In addition to algorithmic techniques such as dynamic programming that can be applied to many problems, there are many tight connections between various problems. This is not surprising when we look at several variations on the same problem, but even problems that on the surface appear very different can be closely related in the following sense. Problem A can be solved with the help of an algorithmic solution to problem B in such a way that we need not make too many calls to the algorithm for B and the additional overhead is acceptable. This implies that if B is efficiently solvable, then A certainly is. Or said in another way: B cannot be efficiently solvable if A is algorithmically difficult. In this way we have used an algorithmic concept (which we will call reduction) to derive the algorithmic difficulty of one problem from the algorithmic difficulty of another. In Chapter 4, this approach will be formalized and practiced with various examples. Of special interest to us will be classes of problems for which every problem can play the role of A in the discussion above and also every problem can play the role of B. Then either all of these problems are efficiently solvable or none of them is. In Chapter 5 we introduce the theory of NP-completeness which leads to the class of problems already discussed in Section 1.1, and to which belong thousands of practically relevant problems that are either all efficiently solvable or all impossible to solve efficiently. The first possibility is equivalent to the property $\mathsf{NP} = \mathsf{P}$ and the second to $\mathsf{NP} \neq \mathsf{P}$. This makes it clear why the $\mathsf{NP} \neq \mathsf{P}$-problem plays such a central role in complexity theory. Now the reductions introduced in Chapter 4 receive their true meaning, since they are used to show that the problems considered there belong to this class of problems. In Chapter 6 we treat the design of such reductions in a more systematic manner.

Chapters 7 and 8 are dedicated to the complexity analysis of difficult problems. We will investigate how one can determine the dividing line between efficiently solvable and difficult variations of a problem. The important special case of approximation problems is handled in Chapter 8. With optimization problems we can relax the demand that we compute an optimal solution if we can be satisfied with an “almost” optimal solution, in which case “almost” must be quantified. For a few problems, results from previous chapters lead relatively easily to results for such approximation problems. To obtain further results by means of reductions, it is necessary to introduce a generalized notion of approximation preserving reduction. Quite a number of approximation problems can be dealt with in this manner; nevertheless, there are also important (and presumably difficult) approximation problems that elude all these methods. Classical complexity theory at this point runs up against an insurmountable obstacle; newer developments are presented in Chapters 11 and 12.

Complexity theory must respond to all developments in the design of efficient algorithms, in particular to the increased use of randomized search heuristics that are not problem specific, such as simulated annealing and genetic algorithms. When algorithms do not function in a problem-specific way, our otherwise problem-specific scenario is no longer appropriate. A more appropriate "black box scenario" is introduced in Chapter 9. In this scenario we have the opportunity to determine the difficulty of problems directly, without any complexity theoretic hypotheses.

Not until the early 1990's when the enormous efforts of many researchers led to the so-called PCP Theorem (probabilistically checkable proofs) was it possible to overcome the previously discussed obstacle in dealing with approximation problems. But even now, over a decade after the discovery of this fundamental theorem, not all of its consequences have been worked out, and the result is still being sharpened. On the other hand, there is still no proof of even the basic variation of the PCP Theorem that can be presented in a textbook. (A treatment of this theorem in a special topics course required 12 two-hour sessions.) Here we will merely describe the path to the PCP Theorem and the central results along this path.

Not until Chapter 10 will we take a brief look at structural complexity theory. We will investigate the inner structure of the complexity class NP and develop a logic-oriented view of NP. From this perspective it is possible to derive generalizations of NP which form the polynomial hierarchy. This will provide a better taxonomy for the classes that depend on randomized algorithms. We also obtain new hypotheses which have a strong basis, although not as strong as that for the $\mathsf{NP} \neq \mathsf{P}$-hypothesis. Later (in Chapters 11 and 14) we will base claims about practically important problems on these hypotheses.

Proofs have the property that they are much easier to verify than to construct. Thus it is possible to understand in a reasonable amount of time an entire textbook, even though the discovery of the results it contains required many researchers and many years. Generally, proofs are not presented in a formal and logically correct manner (supported only by axioms and a few inference rules); instead, authors attempt to convince their readers with less formal arguments. This would be easier to do using interactive communication (which can be better approximated in a lecture than in a textbook or via e-learning). Teachers since the time of Socrates have presented proofs to students in the form of such dialogues. Chapter 11 contains an introduction to interactive proof systems. What does this have to do with the complexity of problems? We measure the complexity in terms of how much communication (measured in bits and communication rounds, in which the roles of speaker and listener alternate) and how much randomization suffice so that someone with unlimited computational resources who knows a proof of some property (e.g., the shortest route in a traveling salesperson problem) can convince someone with realistically limited resources. This original, but seemingly useless game, turns out to have a tight connection to the complexity of the problems we have been discussing. There are even proof dialogues by which the second person can be convinced of a certain property without learning anything new about the proof (so-called zero-knowledge proofs). Such dialogues have an obvious application: The proof could be used as password. The password could then be efficiently checked, without any loss of security, since no information about the password itself need be exchanged. In other words, a user is able to convince the system that she knows her password, without actually providing the password.

After this preparation, the PCP Theorem is treated in Chapter 12 and the central ideas of the proof are discussed. The PCP Theorem will then be used to achieve better results about the complexity of central approximation problems.

Chapter 13 offers a brief look at further themes in classical complexity theory: space-bounded complexity classes, the complexity theoretic classification of context sensitive languages, the Theorems of Savitch and of Immerman and Szelepcsényi, PSPACE-completeness, P-completeness (that is, problems that are efficiently solvable but inherently sequential), and #P-completeness (in which we are concerned with the complexity of determining the number of solutions to a problem).

Chapter 14 treats the complexity theoretic difference between software and hardware. An algorithm (software) works on inputs of arbitrary length while a circuit (hardware) can only process inputs of a fixed length. While there is a circuit solution for every Boolean function in disjunctive normal form (DNF), there are algorithmic problems that are not solvable at all (e.g., the Halting Problem, software verification). The question here will be whether algorithmically difficult problems can have small circuits.

Chapter 15 contains an introduction to the area of communication complexity. Once computer science was defined as the science of processing information. Today the central role of communication is beyond dispute. By means of the theory of communication complexity it has been possible to reduce many very different problems to their common communication core. We will present the fundamental methods of this theory and some example applications.

Boolean (or more general) finite functions clearly play an important role in computer science. There are important models for their computation or representation (circuits, formulas, branching programs – also called binary decision diagrams or BDDs). Their advantage is that they are independent of any short term changes in technology and therefore provide clearly specified reference models. This makes concrete bounds on the complexity of specific problems interesting. Here, too, lower bounds are hard to come by. In Chapter 16, central proof methods are introduced and, together with methods from communication complexity, applied to specific functions.

1.4 Additional Literature

Since we have restricted ourselves in this text to an introduction to complexity theory, and in particular have only treated structural complexity very briefly, we should mention here a selection of additional texts. To begin with, two classic monographs must be cited, each of which has been very influential. The first of this is the general introduction to all areas of theoretical computer science by Hopcroft and Ullman (1979) with the famous cover picture. (An updated version of this text by Hopcroft, Motwani, and Ullman appeared in 2001.) The book by Garey and Johnson (1979) was for many years *the* NP-completeness book, and is still a very good reference due to the large number of problems that it treats. The *Handbook of Theoretical Computer Science* edited by van Leeuwen (1990) provides above all a good placement of complexity theory within theoretical computer science more generally. This book, as well as books by Papadimitriou (1994) and Sipser (1997), treat many aspects of computational complexity. Those looking for a text with an emphasis on structural complexity theory and its specialties are referred to books by Balcázar, Díaz, and Gabarró (1988), Hemaspaandra and Ogihara (2002), Homer (2001), and Wagner and Wechsung (1986). More information about the PCP Theorem can be found in the collection edited by Mayr, Prömel, and Steger (1998). The book by Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela, and Protasi (1999) specializes in approximation problems. Hromkovič (1997) treats aspects of parallel computation and multiprocessor systems especially thoroughly. The complexity of Boolean functions with respect to circuits and formulas is presented by Wegener (1987) and Clote and Kranakis (2002), and with respect to branching programs and BDDs by Wegener (2000). The standard works on communication complexity are Hromkovič (1997) and Kushilevitz and Nisan (1997).