Mathematics and Computation

Mathematics and Computation

A Theory Revolutionizing Technology and Science

Avi Wigderson

Princeton University Press
Princeton and Oxford

Published by Princeton University Press,
41 William Street, Princeton, New Jersey 08540

In the United Kingdom: Princeton University Press,
6 Oxford Street, Woodstock, Oxfordshire OX20 1TR

press.princeton.edu

Library of Congress Control Number: 2018965993
ISBN: 978-0-691-18913-0

British Library Cataloging-in-Publication Data is available

Editorial: Vickie Kearn, Lauren Bucca, and Susannah Shoemaker
Production Editorial: Nathan Carr
Cover design: Sahar Batsry and Avi Wigderson
Production: Jacquie Poirier
Publicity: Matthew Taylor and Kathryn Stevens
Copyeditor: Cyd Westmoreland

This book has been composed in LaTeX

The publisher would like to acknowledge the author of this volume for providing the camera-ready copy from which this book was printed.

Printed on acid-free paper ∞

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Mathematics and Computation

1 Introduction

"$\mathcal{P}$ versus $\mathcal{NP}$ — a gift to mathematics from computer science"

–Steve Smale

Here is just one tip of the iceberg we'll explore in this book: How much time does it take to find the prime factors of a 1,000-digit integer? The facts are that (1) we can't even roughly estimate the answer: it could be less than a second or more than a million years, and (2) practically all electronic commerce and Internet security systems in existence today rest on the belief that it takes more than a million years!

Digesting even this single example begins to illuminate the conceptual revolution of *computational complexity theory*, to which this book is devoted. It illustrates how a pure number theoretic problem, which has been studied for millennia by mathematicians, becomes a cornerstone of a trillion-dollar industry, on which practically all people, companies, and countries crucially depend. Extracting this novel meaning and utility relies on making the above problem precise, and on transforming the informal statement of (2) into a mathematical theorem. This in turn requires formal definitions of such concepts as *algorithm, efficiency, secret*, and *randomness*, among others, including several new notions of *proof*. The difficulty of resolving (the now all-important) challenge (1), namely, proving the hardness of factoring or suggesting alternatives to it, is intimately related to the great conundrum of $\mathcal{P}$ vs. $\mathcal{NP}$. And as a final twist in this plot, a fork appeared in our computational path, by which the answer to (1) may radically depend on whether we allow classical or quantum physics to power our computers. This new possibility has propelled huge investments in academia and industry to attempt to physically realize the potential of quantum computers. It also demands revisiting and redefining the very concepts mentioned above, as well as many physical ones like *entanglement* and *decoherence*, interacting with quantum mechanics and proposing novel ways of testing its foundations.

The book you are reading will explore the mathematical and intellectual aspects of this goldmine. It will explain *computational complexity theory*, the concepts this theory created and revolutionized, and its many connections and interactions with mathematics. In its half-century of existence, computational complexity theory has developed into a rich, deep, and broad mathematical theory with remarkable achievements and formidable challenges. It has forged strong connections with most other mathematical fields and at the same time is having a major practical impact on the *technological revolution* affecting all aspects of our society (of which Internet security and quantum computing above are "mere" examples).

Computational complexity theory is a central subfield of the *theory of computation* (ToC), and is playing a pivotal role in its evolution. This theory stands with the great ones of physics, biology, math, and economics, and is central to a new *scientific revolution* informed by computation. I have devoted the final chapter of this book (which can be read first) to a panoramic overview of ToC. That chapter describes the intellectual supernova that the theory of computing has created and continues to shape. It reveals the broad reach of ToC to all sciences, technology, and society, and discusses its methodology, challenges, and unique position in the intellectual sphere.

Below I review the long history of the interactions of computation and mathematics. I proceed with a short overview of the evolution and nature of computational complexity. I then describe the structure, scope, and the intended audience of this book, followed by a chapter-by-chapter description of its contents.

1.1 On the interactions of math and computation

The Theory of Computation is the study of the formal foundations of computer science and technology. This dynamic and rapidly expanding field straddles mathematics and computer science. It has benefited tremendously from the very different characters, motivations, and traditions of these parent disciplines. Both sides naturally emerged from its birth, in the “big bang” of Turing’s seminal 1936 paper [Tur36], “On computable numbers, with an application to the Entscheidungsproblem.” This is a paper written by a PhD student, in the area of mathematical logic, which, combined with its long title, might seem to condemn it to obscurity. However, with Turing’s incredible clarity of vision, exposition, and motivation, it is an inspiring model in mathematical modeling with singular impact. This paper formally defined an *algorithm* in the form of what we call today the “Turing machine.” On one hand, the Turing machine is a formal mathematical model of computation, enabling for the first time the rigorous definition of computational tasks, the algorithms to solve them, and the basic resources these require (in particular, allowing Turing to prove that very basic tasks are uncomputable). On the other hand, the extremely elegant definition of the Turing machine allowed its simple, logical design to be readily implemented in hardware and software, igniting the computer revolution.

These theoretical and practical aspects form the dual nature of ToC and strongly influenced the field and its evolution. On the mathematical side, the abstract notion of computation revealed itself as an extremely deep and mysterious notion that illuminates other, often well-studied concepts in a new light. In pursuing the abstract study of computation, ToC progresses like any other mathematical field. Its researchers prove theorems and follow mathematical culture to generalize, simplify, and create variations, following their instincts based on esthetics and beauty. On the practical side, the universal applicability of automated computation fueled the rapid development of computer technology, which now dominates our life. The interaction between theory and practice never stops. The evolving world of computer science and industry continuously creates new types and properties of computation, which need theoretical modeling and understanding, and directly impacts the mathematical evolution of ToC. Conversely, the ideas, models, and techniques generated there feed back into the practical world. Besides technology, more recent and growing sources of external influence on ToC are nature and science. Many natural processes can (and should) be understood as information processes and demand similar computational understanding. Here again theoretical modeling, techniques, and new theoretical questions feed back to suggest experiments and better understanding of scientific data. Much more on these connections is discussed in Chapter 20.

Needless to say, mathematics and computation did not meet in 1936 for the first time; they have been tied to each other from the dawn of human civilization. Indeed, ancient mathematics developed primarily from the need to compute, be it for predicting natural phenomena of all types, managing crops and livestock, manufacturing and building, trading commodities, and planning for the future. Devising representations of numbers and developing efficient methods for performing arithmetic on them were thus central. More generally, in a very fundamental way, a mathematical understanding could solve any practical problem only *through* a computational process applied to the data at hand. So, even though algorithms were formally defined only in the twentieth century, mathematicians and scientists continuously devised, described, and used better and better algorithms (albeit informally explained and rarely analyzed) for the computations required to draw conclusions from their theories. Examples abound, and we list just a few highlights. Euclid, working ca. 300 BCE, devised his fast GCD[1] algorithm to bypass the need to laboriously factor integers when simplifying fractions. Euclid's famous 13-volume *Elements*, the central math text for many centuries, contains dozens of other algorithms to compute numerical and geometric quantities and structures. In the same era, Chinese mathematicians compiled *The Nine Chapters on the Mathematical Art*, which contains many computational methods, including "Gaussian elimination" (for solving systems of linear equations). In the ninth century, al-Khwārizmī (after whom "algorithm" is named) wrote his books *Compendious Book on Calculation by Completion and Balancing* and *On the Hindu Art of Reckoning*. These books respectively expound on everything known up to that time about algorithms for algebraic and arithmetic problems, such as solving quadratic equations and linear systems, and performing arithmetic operations in the decimal system. The very reason that the decimal system survived as the dominant way to represent numbers is the usefulness of these efficient algorithms for performing arithmetic on arbitrarily large numbers so represented.

The "modern era" has intensified these connections between math and computation. Again, examples. During the Renaissance, mathematicians found *formulas*, the most basic computational recipe, for solving cubic and quartic equations via radicals.[2] Indeed, famous competitions between Tartaglia, Piore, Ferrari, and others in the early 1500s were all about who had a faster algorithm for solving cubic equations. The Abel-Ruffini theorem that the quintic equation has no such formula is perhaps the earliest *hardness result*: It proves the non existence of an algorithm for a concrete problem in a precise computational model. Newton's *Principia Mathematica* is a masterpiece not only of grand scientific and mathematical theories; it is also a masterpiece of algorithms for computing the predictions of these theories. Perhaps the most famous and most general is "Newton's method" for approximating the roots of real polynomials of arbitrary degree (practically bypassing the Abel-Ruffini obstacle mentioned above). The same can be said about Gauss' magnum opus, *Disquisitiones Arithmeticae*—it is full of algorithms and computational methods. One famous example (published after his death), is his discovery[3] of the fast Fourier transform (FFT), the central algorithm of signal processing, some 150 years before its "official" discovery by J. W. Cooley and J. W. Tukey. Aiming beyond concrete problems, Leibniz, Babbage, Lovelace, and others pioneered explicit attempts to design, build, and program general-purpose computational devices. Finally, Hilbert dreamed of resting all of mathematics on computational foundations, seeking a "mechanical procedure" that would (in principle) determine all mathematical truths. He believed that truth and proof coincide (i.e., that every true statement has a valid proof), and that such proofs can be found automatically by such a computational procedure. The quest to formalize Hilbert's program in math-

[1]The GCD (greatest common divisor) problem is to compute the largest integer that evenly divides two other integers.

[2]Namely, using arithmetic operations and taking roots.

[3]For the purpose of efficiently predicting the orbits of certain asteroids.

ories predated (and indeed enabled) significant technological advances.[5] In other cases, these theories formed the basis of interactions with other sciences.

Thus, starting with the goal of understanding what can be efficiently *computed*, a host of natural long-term goals of deep conceptual meaning emerged. What can be efficiently learned? What can be efficiently proved? Is verifying a proof much easier than finding one? What does a machine know? What is the power of randomness in algorithms? Can we effectively use natural sources of randomness? What is the power of quantum mechanical computers? Can we utilize quantum phenomena in algorithms? In settings where different computational entities have different (possibly conflicting) incentives, what can be achieved jointly? Privately? Can computers efficiently simulate nature, or the brain?

The study of efficient computation has created a powerful methodology with which to investigate such questions. Here are some of its important principles, which we will see in action repeatedly, and whose meaning will become clearer as we proceed in this book. *Computational modeling*: uncover the underlying basic operations, information flow, and resources of processes. *Efficiency*: attempt to minimize resources used and their trade-offs. *Asymptotic thinking*: study problems on larger and larger objects, as structure often reveals itself in the limit. *Adversarial thinking*: always prepare for the worst, replacing specific and structural restrictions by general, computational ones—such more stringent demands often make things simpler to understand! *Classification*: organize problems into (complexity) classes according to the resources they require. *Reductions*: ignore ignorance, and even if you can't efficiently solve a problem, assume that you can, and explore which other problems it would help solve efficiently. *Completeness*: identify the most difficult problems in a complexity class.[6] *Barriers*: when stuck for a long time on a major question, abstract all known techniques used for it so far and try to formally argue that they will not suffice for its resolution.

These principles work extremely well with one another, and in surprisingly diverse settings, especially when applied at the right level of abstraction (which I believe has been indeed cleverly chosen, repeatedly). This methodology allowed ToC to uncover hidden connections among different fields and create a beautiful edifice of structure, a remarkable order in a vast collection of notions, problems, models, resources, and motivations. While many of these principles are in use throughout mathematics and science, I believe that the disciplined, systematic use that has become ingrained in the culture of computational complexity—especially of problem classification via (appropriate) reductions and completeness—has great potential to further enhance other academic fields.

The field of Computational complexity is extremely active and dynamic. While I have attempted to describe, besides the fundamentals, some of the recent advances in most areas, I expect the state of the art will continue to expand quickly. Thus, some of the open problems will become theorems, new research directions will be created, and new open problems will emerge. Indeed, this has happened repeatedly in the few years it has taken me to write this book.

[5]The best example is cryptography, which in the 1980s was purely motivated by a collection of fun intellectual challenges, like playing poker over the telephone, but developed into a theory that enabled the explosive growth of the Internet and e-commerce. (This will be discussed in Chapter 18.)

[6]Namely, those which all other problems in the class reduce to in the sense mentioned above.

1.3 The nature, purpose, and style of this book

Computational complexity theory has been my intellectual (and social) home for almost 40 years. During this period, I have written survey articles and have given many more survey lectures on various aspects of of this field. This book grew out of these expositions and out of the many other aspects I had planned to write and talk about, but had never gotten around to. Indeed, breadth of scope is an important goal of the book. Furthermore, as in my lectures and surveys, so in this book, I try to explain not only what we do, but also why we do it, why is it so important, and why is it so much fun!

The book explores the foundations and some of the main research directions of computational complexity theory, and their many interactions with other branches of mathematics. The diversity of these computational settings is revealed when we discuss the contents of every chapter below. For every research area, the book focuses on the main aspects of computation it attempts to *model*. The book presents the notions, goals, results, and open problems for each area in turn, all from a conceptual perspective, providing ample motivation and intuition. It describes the history and evolution of ideas leading to different notions and results and explains their meaning and utility. It also highlights the rich tapestry of (often surprising and unexpected) connections among the different subareas of computational complexity theory; this unity of the field is an important part of the field's success.

To highlight the conceptual perspective, the material is generally presented at a high level and in somewhat informal fashion. Almost no proofs are given, and I focus on discussions of general proof techniques and key ideas at an informal level. Precise definitions, theorem statements, and of course detailed proofs for many topics discussed can be found in the excellent textbooks on computational complexity [Pap03, Gol08, AB09, MM11]. Also, for historical reasons and greater detail, I provide many references to original papers, more specialized textbooks, and survey articles in every chapter.

1.4 Who is this book for?

I view this book as useful to several audiences in several different ways.

- First, it is an invitation to advanced undergraduates and beginning graduate students in math, computer science, and related fields, to find out what this field is about, get excited about it, and join it as researchers.

- Second, it should serve graduate students and young researchers working in some area of CS theory to broaden their views and deepen their understanding about other parts of the field and their interconnectivity.

- Third, computer scientists, mathematicians, researchers from other fields, and motivated nonacademics can get a high-level view of computational complexity, its broad scope, achievements, and ambitions.

- Last but not least, educators in the field can use different parts of the book for planning and supplementing a variety of undergraduate and graduate courses. I hope that the conceptual view of the field, its methodology, its unity, and the beauty and excitement of its achievements and challenges that I have labored to present here will help inform and animate these courses.

Different chapters may require somewhat different backgrounds, but the introduction to each aims to be welcoming and gentle. The first two chapters and the last one should be accessible to most of the above mentioned audiences.

Let me conclude this section with a piece of advice that may be useful to some readers. It can be tempting to read this book quickly. Many parts of the book require hardly any specific prior knowledge and rely mostly on the mathematical maturity needed to take in the definitions and notions introduced. Hopefully, the story telling makes the reading even easier. However, the book (like the field itself) is conceptually dense, and in some parts the concentration of concepts and ideas requires, I believe, slowing down, rereading, and possibly looking at a relevant reference to clarify and solidify the material and its meaning in your mind.

1.5 Organization of the book

Below I summarize the contents of each chapter in the book. Naturally, some of the notions mentioned below will only be explained in the chapters themselves. After the introductory Chapters 2 and 3, the remaining ones can be read in almost any order. Central concepts (besides computation itself) that sweep across several chapters include *randomness* (Chapters 7–10), *proof* (Chapters 3, 6, and 10) and *hardness* (Chapters 5, 6, and 12).

Different partitions can be made around the focus of different sets of chapters. Chapters 2–12 focus mostly on one computational resource, *time*, namely, the number of steps taken by a single machine (of various types) to solve a problem. Chapters 14–19 (as well as 10) deal with other resources and with more complex computational environments, in which interactions take place among several computational devices. Finally, while mathematical *modeling* is an important part of almost every chapter, it is even more so for the complex computational environments we'll meet in Chapters 15–19, where modeling options, choices, and rationales are discussed at greater length. Chapters 13 and 20 are standalone surveys, the first on concrete interactions between math and computational complexity and the second on the Theory of Computation. Here are brief descriptions of each chapter (the headlines below may differ from the chapter title).

Prelude: computation and mathematical understanding

Chapter 2 is the prelude to the arrival of computational complexity, starting with the formalization of the notion of *algorithm* as the Turing machine. We discuss basic computational problems in mathematics and algorithms for them. We then explore the relevance of the boundary between decidable and undecidable problems about classes of mathematical structures, in the hope of completely understanding them.

Computational complexity 101 and the $\mathcal{P}$ vs. $\mathcal{NP}$ question

Chapter 3 introduces the basic concepts of computational complexity: decision problems, time complexity, polynomial vs. exponential time, efficient algorithms, and the class $\mathcal{P}$. We define efficient verification and the class $\mathcal{NP}$, the first computational notion of proof. We proceed with efficient reductions between problems and the notion of completeness. We then introduce $\mathcal{NP}$-complete problems and the $\mathcal{P}$ vs. $\mathcal{NP}$ question. Finally, we discuss related problems and complexity classes. For all these notions, I motivate some of the choices made and explain their importance for computer science, math, and beyond.

Different types of computational problems and complexity classes

Chapter 4 introduces new types of *questions* one can ask about an input beyond classification, including counting, approximation, and search. We also move from worst-case performance of algorithms to success or failure on average, and discuss the related notions of one-way and trap-door functions underlying cryptography. I explain how the methodology developed in the previous chapter leads to other complexity classes, reductions, and completeness. This begins to paint the richer structure organizing problems above, below, and "around" $\mathcal{NP}$.

Hardness and the difficulties it presents

Chapter 5 discusses *lower bounds*—the major challenge of proving that $\mathcal{P}$ is different from $\mathcal{NP}$, and more generally, proving that some natural computational problems are hard. Central to this section is the model of Boolean circuits, a "hardware" analog of Turing machines. We review the main techniques used for lower bounds on restricted forms of circuits and Turing machines. We also discuss the introspective barrier results, explaining why these techniques seem to fall short of the real thing: lower bounds for general models.

How deep is your proof?

Chapter 6 introduces *proof complexity*, another view of the basic concept of *proof*. Proof complexity applies computational complexity methodology to quantify the difficulty of proving natural theorems. We describe a variety of propositional proof systems—geometric, algebraic, and logical—all capturing different ways and intuitions of making deductions for proving natural tautologies. I explain the ties among proof systems, algorithms, circuit complexity, and the *space: the final frontier* problem. We review the main results and challenges in proving lower bounds in this setting.

The power and weakness of randomness for algorithms

Chapter 7 reviews using randomness to enhance the power of algorithms. We define probabilistic algorithms and the class $\mathcal{BPP}$ of problems they solve efficiently. We discuss such problems (among numerous others) for which no fast deterministic algorithms are known, which suggests that randomness is powerful. However, this intuition may be an illusion. We next introduce the fundamental notions of *computational pseudo-randomness, pseudo-random generators*, the *hardness vs. randomness paradigm*, and *de-randomization*. These ideas suggest that randomness is weak, at least assuming hardness statements like $\mathcal{P} \neq \mathcal{NP}$. The chapter concludes with a discussion of the evolution and sources of these ideas, their surprising consequences, and their impact beyond the power of randomness.

Is π random?

Chapter 8 covers "random looking" deterministic structures. We discuss "abstract" pseudo-randomness, a general framework that extends computational pseudo-randomness and accommodates a variety of natural problems in mathematics and computer science. We define pseudo-random properties and discuss the question of deterministically finding pseudo-random objects. We will see how the $\mathcal{P}$ vs. $\mathcal{NP}$

problem, the Riemann hypothesis, and many other problems that naturally fall into this framework can be viewed as questions about pseudo-randomness. Finally, we discuss the structure vs. pseudo-randomness dichotomy, a paradigm for proving theorems in a variety of areas, and examine the scope of this idea.

Utilizing the unpredictability of the weather, stock prices, quantum effects, etc.

Chapter 9 discusses weak random sources, a mathematical model of some natural phenomena that seem to be somewhat unpredictable but may be far from a perfect stream of random bits. We raise the question of if and how probabilistic algorithms can use such weak randomness and define the main object used to answer this question—the randomness extractor. We then explore the evolution of ideas leading to efficient constructions of extractors, and the remarkable utility of this pseudo-random object for other purposes.

Interactive proofs: teaching students with coin tosses

Chapter 10 addresses yet again, proofs this time concerning the impact of introducing randomness and interaction into the definition of proofs. These new notions of proofs give rise to new complexity classes, like $\mathcal{IP}$ and $\mathcal{PCP}$, and their surprising characterization in terms of standard complexity classes. We explore how this setting allows new types of proofs, like zero-knowledge proofs and spot-checking proofs, and their implications for cryptography and hardness-of-approximation.

Schrödinger's laptop: algorithms meet quantum mechanics

Chapter 11 introduces quantum computing: algorithms endowed with the ability to use quantum mechanical effects in their computation. We discuss important algorithms for this theoretical model, how they motivated a large-scale effort to build quantum computers, and the status of this effort. We extend the idea of proofs again by using this notion and discuss complete problems for quantum proofs. This turns out to connect directly to quantum Hamiltonian dynamics, a central area in condensed matter physics, and we explore some of the interactions between the fields. Finally, we discuss the power of interactive proofs to answer the basic question: Is quantum mechanics falsifiable?

Arithmetic complexity: plus and times revisited

Chapter 12 leaves the Boolean domain and introduces the model of arithmetic circuits, which use arithmetic operations to compute polynomials over (large) fields. We review the main results and open problems in this area, relating it to the study of Boolean circuits. We exposit Valiant's arithmetic complexity theory, its main complexity classes $\mathcal{VP}$ and $\mathcal{VNP}$, complete problems, the determinant, and permanent polynomials. We discuss a recent approach to solve the $\mathcal{VP}$ vs. $\mathcal{VNP}$ problem via algebraic geometry and representation theory. We also survey a collection of restricted models for which strong lower bounds are known.

All the chapters so far focus on one primary computational resource: *time* (or more generally, the number of elementary operations). Before broadening our scope, we take a break with an interlude.

to date version. In some cases these are journal papers (which take longest to appear), some are conference publications (which appear faster), and in rare cases only electronic versions exist (which are instantaneous). Also, when I call a result "recent", please note that this is relative to the publication date of the book.

- **Footnotes** There are *many* footnotes in this book. In almost all cases, they are designed to enrich, elaborate or further explain something. However, the text itself should be self-contained without them. Therefore you can safely skip footnotes; this will rarely affect understanding.
- **iff** The shorthand *iff* will be used throughout to mean "if and only if".
- **Asymptotic notation** Crucial to most chapters is the following asymptotic notation, relating "growth in the limit" of functions on the integers. Let f, g be two integer functions. Then we denote:
  - $f = O(g)$, if for some positive constant C, for all large enough n, $f(n) \leq C \cdot g(n)$.
  - $f = \Omega(g)$, if for some positive constant c, for all large enough n, $f(n) \geq c \cdot g(n)$.
  - $f = \Theta(g)$, if both $f = O(g)$ and $f = \Omega(g)$ hold.
  - $f = o(g)$, if $f(n)/g(n)$ tends to zero as n tends to infinity.

  We say that an integer function f grows (at most) *polynomially* if there are constants A, c such that for all n, $f(n) \leq An^c$.
  We say that f grows (at most) *exponentially* if there are constants A, c such that for all n, $f(n) \leq A \exp(n^c)$.

2 *Prelude*: Computation, undecidability, and limits to mathematical knowledge

This short section will briefly recount the history of ideas leading the the birth of computational complexity theory.

Mathematical classification problems Which mathematical structures can we hope to understand? Consider any particular class of mathematical objects and any particular relevant property. We seek to *understand* which of the objects have the property and which do not. Examples of this very general *classification problem* include the following[1]:

1. Which Diophantine equations have solutions?
2. Which knots are unknotted? (see Figure 1)
3. Which planar maps are 4-colorable? (see Figure 2)
4. Which theorems are provable in Peano arithmetic?
5. Which pairs of smooth manifolds are diffeomorphic?
6. Which elementary statements about the reals are true?
7. Which elliptic curves are modular?
8. Which dynamical systems are chaotic?

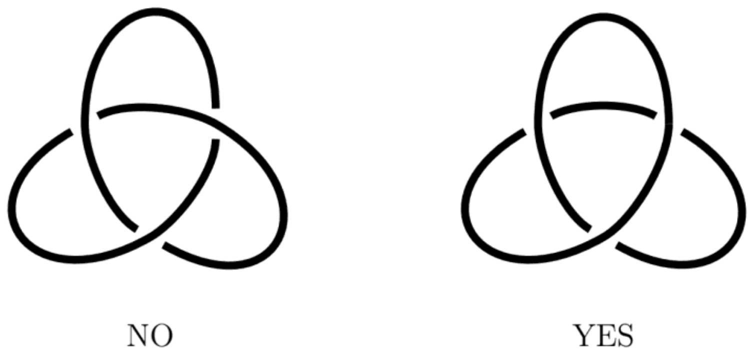
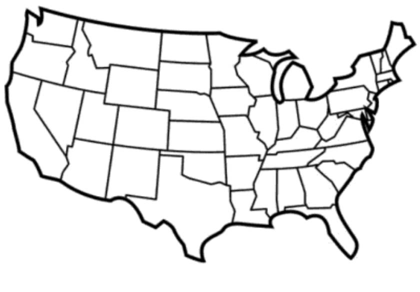


Figure 1. Instances of problem 2 and their classification. The left is a diagram of the Trefoil knot and the right is one of the Unknot.

[1]It is not essential that you understand every mathematical notion mentioned below. If curious, reading a Wikipedia-level page should be more than enough for our purposes here.



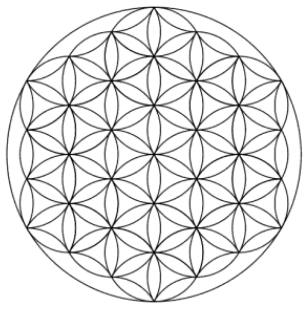


Figure 2. Instances of problem 3 and their classification. Both maps are 4-colorable.

Understanding and algorithms A central question is what we mean by *understanding.* When are we satisfied that our classification problem has been reasonably solved? Are there problems like these that we can never solve? A central observation (popularized mainly by David Hilbert) is that "satisfactory" solutions usually provide (explicitly or implicitly) *mechanical procedures*, which when applied to an object, determine (in finite time) whether it has the property. Hilbert's problems 1 and 4 above were stated, it seems, with the expectation that the answer would be positive; namely, that mathematicians would be able to understand them in this very computational sense.

So, Hilbert identified mathematical knowledge with computational access to answers, but never formally defined computation. This task was taken up by logicians in the early twentieth century and was met with resounding success in the 1930s. The breakthrough developments by Gödel, Turing, Church, and others led to several quite different formal definitions of computation, which happily turned out to be identical in power. Of all, Turing's 1936 paper [Tur36] was most influential. *Indeed, it is easily the most influential math paper in history.* We already mentioned in Section 1.1 that in this extremely readable paper, Turing gave birth to the discipline of computer science and ignited the computer revolution, which radically transformed society. Turing's model of computation (quickly named a *Turing machine*) became one of the greatest intellectual inventions ever. Its elegant and simple design on the one hand, and its universal power (elucidated and exemplified by Turing) on the other immediately led to implementations, and the rapid progress since then has forever changed life on Earth. This paper serves as one of the most powerful demonstrations of how excellent theory predates and enables remarkable technological and scientific advances. But on top of all these, Turing's paper also resolved problems 1 and 4 above! In the negative!! Let us see how.

With the Turing machine, we finally have a rigorous definition of computation, giving formal meaning to Hilbert's questions. It allows proving mathematical theorems about what "mechanical procedures" can and cannot do! *Turing defined an **algorithm** (also called a* decision procedure*) to be a Turing machine (in modern language, simply a computer program) that halts on every input in finite time.* So, algorithms compute functions, and being finite objects themselves, one immediately sees from a Cantor-like diagonalization argument that some (indeed almost all) functions are *not* computable by algorithms. Such functions are also called *undecidable*; they have no decision procedure. But Turing went further and showed that specific, natural functions, like Hilbert's Entscheidungsproblem (4) above were undecidable (this was independently proved by Church as well). Turing's elegant 1-page proof adapts a Gödelian self-reference argument on a Turing machine.[2] These demonstrate powerfully the mathematical value of Turing's basic computational model.

Decidability and undecidability Turing thus shattered Hilbert's first dream. Problem 4 being undecidable means that we will never understand in general, in Hilbert's sense, which theorems are provable (say, in Peano arithmetic): No algorithm can discern provable from unprovable theorems. It took 35 more years to do the same to Hilbert's problem 1. Its undecidability (proved by Davis, Putnam, Robinson, and Mattiasevich in 1970) says that we will never understand in this way polynomial equations over integers: No algorithm can discern solvable from unsolvable ones.

A crucial ingredient in those (and all other undecidability) results is showing that each of these mathematical structures (Peano proofs, integer polynomials) can *"encode computation"* (in particular, these seemingly static objects encode a dynamic process). This is known today to hold for many different mathematical structures in algebra, topology, geometry, analysis, logic, and more, even though *a priori* the structures studied seem to be completely unrelated to computation. This ubiquity makes every mathematician a potential computer scientist in disguise. We shall return to refined versions of this idea later.

Naturally, such negative results did not stop mathematical work on these structures and properties—they merely suggested the study of interesting subclasses of the given objects. Specific classes of Diophantine equations were understood much better, for example, Fermat's Last Theorem and the resolution of problem (7) regarding the modularity of elliptic curves. The same holds for restricted logics for number theory (e.g. Presburger arithmetic).

The notion of a decision procedure (or algorithm) as a minimal requirement for understanding a mathematical problem has also led to direct positive results. It suggests that we should look for a decision procedure as *a means*, or as the *first step* for understanding a problem. With this goal in mind, Haken [Hak61] showed how knots can be understood in this sense, designing a decision procedure for problem (2), determining knottedness. Similarly Tarski [Tar51] showed that closed real fields can be thus understood, designing a decision procedure for problem (6). Naturally, significant *mathematical, structural* understanding was needed to develop these algorithms. Haken developed the theory of *normal surfaces* and Tarski invented *quantifier elimination* for their algorithms; in both cases, these ideas and techniques became cornerstones of their respective fields.

These important examples, and many others like them, only underscore what has been obvious for centuries: mathematical and algorithmic understanding are strongly related and often go hand in hand, as discussed at length in the introduction. And what was true in previous centuries is still true in this one: The language of algorithms is compatible with and actually generalizes the language of equations and formulas (which are special cases of algorithms), and is a powerful language for

[2]As a side bonus, a similar argument gives a short proof of Gödel's incompleteness theorem, a fact which for some reason is still hidden from many undergraduates taking logic courses.

understanding and explaining complex mathematical structures.

The many decision procedures developed for basic mathematical classification problems, such as Haken's and Tarski's solutions to problems (2) and (6), respectively, demonstrate this notion of algorithmic understanding *in principle*. After all, what they guarantee is an algorithm that will deliver the correct solution in *finite* time. Should this satisfy us? Finite time can be very long, and it is hard to distinguish a billion years from infinity. This is not just an abstract question for the working mathematician, as we recounted in Section 1.1. Indeed, both Haken's and Tarski's original algorithms were extremely slow, and computing their answer for objects of moderate size may indeed have required a billion years. Can this be quantified?

Efficient algorithms and computational complexity This viewpoint suggests developing and using a computational yardstick and measuring the quality of understanding by the quality of the algorithms providing it. Indeed, we argue that better mathematical understanding of given mathematical structures often goes hand in hand with better algorithms for classifying their properties. Formalizing the notions of algorithms' resources (especially *time*) and their efficient use is the business of computational complexity theory, the subject of this book, which we shall start developing in the next section. But before we do, I would like to use the set of problems above to highlight a few other issues, which we will not discuss further here.

First, the *representation* of objects which algorithms process becomes important! One basic issue raised by most of the problems above is the contrast between continuity in mathematics and discreteness of computation. Algorithms manipulate finite objects (like bits) in discrete time steps. Knots, manifolds, dynamical systems, and the like are continuous objects. How can they be described to an algorithm and processed by it? As many readers will know, the answers vary, but finite descriptions exist for all. For example, we use knot diagrams for knots, triangulations for manifolds, and symbolic descriptions or successive approximations for dynamical systems. It is these *discrete* representations that are indeed used in algorithms for these (and other) *continuous* problems (as, e.g., Haken's algorithm demonstrates). Observe that this has to be the case; every continuous object we humans will ever consider has discrete representations! After all, math textbooks and papers are finite sequences of characters from a finite alphabet, just like the input to Turing machines. And we, their readers, would never be able to process and discuss them otherwise. All this does not belittle the difficulties that may arise when seeking representations of continuous mathematical structures that would be useful for description, processing, and discussion—instead, it further illustrates the inevitable ties between mathematics and computation.

Let me demonstrate that algorithmic efficiency may crucially depend on representation even for simple *discrete* structures. Where would mathematics (and society) be if we continued using *unary* encodings of integers, or even Roman numerals? The *great* invention of the decimal encoding (or more generally, the *positional* number system) was motivated by, and came equipped with, efficient algorithms for arithmetic manipulation! And this is just one extremely basic example.

Problem 3 on the 4-colorability of planar maps points to a different aspect of the interaction of computation and mathematics. Many readers will know that problem (3) has a very simple decision procedure: Answer "yes" on every input.

this chapter and be formalized in Section 3.9. To get there, we need to develop the language and machinery that yield such surprising results.

Representation issues We start by discussing (informally and by example) how such varied complex mathematical objects can be described in finite terms, eventually as a sequence of bits. Often there are several alternative representations, and typically it is simple to convert one to the other. Let us discuss the representation of inputs in these three problems.

For problem (1′) consider first the set of all equations of the form $Ax^2+By+C=0$ with integer coefficients A, B, C. A finite representation of such equations is obvious—the triple of coefficients (A, B, C), say with each integer written in binary notation. Given such a triple, the decision problem is whether the corresponding polynomial has a positive integer root (x, y). Let $2DIO$ denote the subset of triples for which the answer is YES.

Finite representation of inputs to problem (2′) is tricky but still natural. The inputs consist of a 3-dimensional manifold M, a knot K embedded in it, and an integer G. A finite representation can describe M by a triangulation (a finite collection of tetrahedra and their adjacencies). The knot K will be described as a closed path along edges of the given tetrahedra. Given a triple (M, K, G), the decision problem is whether a surface that K bounds has genus at most G. Let $KNOT$ denote the subset for which the answer is YES.

Finite representation of inputs to problem (3′) is nontrivial as well. Let us discuss not maps but instead graphs, in which vertices represent the countries and edges represent adjacency of countries (this view is equivalent; for a planar map, its graph is simply its dual map). To describe a graph (in a way that makes its planarity evident), one elegant possibility is to use a simple and beautiful theorem of Fáry [Fár48] (discovered independently by others and which has many proofs). It states that every planar graph has a *straight line* embedding in the plane (with no edges crossing). So, the input can be a set V of coordinates of the vertices (which can in fact be small integers) and a set E of edges, each a pair of elements from V. Let *3COL* be the subset of those inputs (V, E) describing a 3-colorable map.

Any finite object (integers, tuples of integers, finite graphs, finite complexes, etc.) can be represented naturally by binary sequences over the alphabet $\{0, 1\}$, and this is how they will be described as inputs to algorithms. As discussed above, even continuous objects like knots have finite descriptions and so can be described this way as well.[2] We will not discuss here subtle issues like whether objects have unique representations or whether every binary sequence should represent a legal object. It suffices to say that in most natural problems, this encoding of inputs can be chosen such that these are not real issues. Moreover, going back and forth between the object and its binary representation is simple and efficient (a notion to be formally defined below).

Consequently, let $\mathbf{I}$ denote the set of all finite binary sequences, and regard it as the set of inputs to all our classification problems. Indeed, every subset of $\mathbf{I}$ defines a classification problem. In this language, given a binary sequence $x \in \mathbf{I}$, we may interpret it as a triple of integers (A, B, C) and ask whether the related equation is in $2DIO$. This is problem (1′). We can also interpret x as a triple (M, K, G) of

[2]Theories of algorithms which have continuous inputs (e.g., real or complex numbers) have been developed, for example, in [BCSS98, BC06], but will not be discussed here.

manifold, knot, and integer, and ask whether it is in the set *KNOT*. This is problem $(2')$, and the same can be done with $(3')$.

Reductions Theorem 3.1 states that there are *simple* translations (in both directions) between solving problem $(1')$ and problem $(2')$. More precisely, it provides efficiently computable functions $f, h\colon \mathbf{I} \to \mathbf{I}$ performing these translations:

$(A, B, C) \in 2DIO$ iff $f(A, B, C) \in KNOT$,

and

$(M, K, G) \in KNOT$ iff $h(M, K, G) \in 2DIO$.

Thus, an efficient algorithm to solve one of these problems immediately implies a similar one for the other. Putting it more dramatically, if we have gained enough understanding of topology to solve, say, the knot genus problem, it means that we automatically have gained enough number theoretic understanding for solving these quadratic Diophantine problems (and vice versa).

The translating functions f and h are called *reductions*. We capture the *simplicity* of a reduction in *computational* terms, demanding that it will be *efficiently* computable.

Similar pairs of reductions exist between the map 3-coloring problem and each of the other two problems. If sufficient understanding of graph theory leads to an efficient algorithm to determine whether a given planar map is 3-colorable, similar algorithms follow for both *KNOT* and *2DIO*. And vice versa—understanding either of them will similarly resolve 3-coloring. Note that this positive interpretation of the equivalence paints all three problems as equally "accessible." But the flip side says that they are also equally intractable: If any one of them lacks such an efficient classification algorithm, so do the other two. Indeed, with the better understanding of these equivalences today, it seems more likely that the second interpretation is right: These problems are all hard to understand.

When teaching this material in class, or in lectures to unsuspecting audiences, it is always fun to watch listeners' amazement at these remarkably strong and unexpected connections between such remote problems. I hope it has a similar impact on you. But now it is time to dispel the mystery and explain the source of these connections. Here we go.

3.2 Efficient computation and the class $\mathcal{P}$

Efficient algorithms are the engine which drives an ever-growing part of industry and economy, and with it your everyday life. These jewels are embedded in most devices and applications you use daily. In this section we abstract a mathematical notion of efficient computation, polynomial-time algorithms. We motivate it and give examples of such algorithms.

In all that follows, we focus on asymptotic complexity. Thus, e.g., we care neither about the time it takes to factor the number $2^{67} - 1$ (as much as Mersenne cared about it), nor about the time it takes to factor all 67-bit numbers, but rather about the asymptotic behavior of factoring n-bit numbers, as a function of the input length n. The asymptotic viewpoint is inherent to computational complexity theory, and we shall see in this book that it reveals structure which would be obscured by finite, precise analysis. We note that the dependence on input size does not exist in Computability theory, where algorithms are simply required to halt in finite time.

However, much of the methodology of these fields was imported to computational complexity theory—complexity classes of problems, reductions between problems and complete problems, all of which we shall meet.

Efficient computation (for a given problem) will be taken to be one whose run-time on any input of length n is bounded by a *polynomial* function in n.

Recall that $\mathbf{I}$ denotes the set of all binary sequences of all lengths. Let $\mathbf{I}_n$ denote all binary sequences in $\mathbf{I}$ of length n, namely $\mathbf{I}_n = \{0,1\}^n$.

Definition 3.2 (The class $\mathcal{P}$)**.** A function $f\colon \mathbf{I} \to \mathbf{I}$ is in the class $\mathcal{P}$ if there is an algorithm computing f and positive constants A, c, such that for every n and every $x \in \mathbf{I}_n$, the algorithm computes $f(x)$ in at most An^c steps (namely, elementary operations).

Note that the definition applies in particular to Boolean functions (whose output is $\{0,1\}$), which capture classification problems (often called "decision problems"). We will abuse notation and sometimes think of $\mathcal{P}$ as the class containing *only* these classification problems. Observe that a function with a long output can be viewed as a sequence of Boolean functions, one for each output bit.

This important definition, contrasting polynomial growth with (brute force) exponential growth, was put forth in the late 1960s by Cobham [Cob65], Edmonds [Edm65b, Edm66, Edm67a], and Rabin [Rab67]. These researchers, coming from different areas and motivations, all attempted to formally delineate *efficient* from just *finite* algorithms. Edmonds's papers in particular supply some ingenious polynomial time algorithms to natural optimization problems. Of course, nontrivial polynomial-time algorithms were discovered earlier, long before the computer age. Many were discovered by mathematicians who needed efficient methods to calculate (by hand). The most ancient and famous example is of course Euclid's GCD (greatest common divisor) algorithm mentioned in Chapter 1, which was invented to bypass the need to factor integers when computing their largest common factor.

Two major choices must be made in selecting $\mathcal{P}$ to model the class of efficiently computable functions, which are often debated and certainly demand explanation. One is the choice of *polynomial* as the bound on time in terms of input length, and the second is the choice of *worst-case* requirement (namely, that this time bound holds for all inputs). We discuss the motivation and importance of these two choices below. However, it is important to stress that these choice are not dogmatic: computational complexity theory has considered and investigated numerous other alternatives to these choices. These include many finer grained bounds on efficiency other than polynomial, and many different notions of average case and input-dependent measures replacing the worst-case demands. Some of theses will be discussed later in the book. Still, the initial choices above were extremely important in the early days of computational complexity, revealing beautiful structure that would become the solid foundation for the field, establish its methodology, and guide the subsequent study of finer and more diverse alternatives.

3.2.1 Why polynomial?

The choice of polynomial time to represent efficient computation seems arbitrary. However, this particular choice has justified itself over time from many points of view. I list some important justifications.

Polynomials typify "slowly growing" functions. The closure of polynomials under addition, multiplication, and composition preserves the notion of efficiency under natural programming practices, such as using two programs in sequence, or using one as a subroutine of another. This choice removes the necessity of describing the computational model precisely (e.g., it does not matter whether we allow arithmetic operations only on single digits or on arbitrary integers, since long addition, subtraction, multiplication, and division have simple polynomial-time algorithms taught in grade school). Similarly, we need not worry about data representation: one can efficiently translate between essentially any two natural representations of a set of finite objects.

From a practical viewpoint, a running time of, say, n^2 is far more desirable than n^{100}, and of course linear time is even better. Indeed even the constant coefficient of the polynomial running time can be crucial for real-life feasibility of an algorithm. However, there seems to be a "law of small numbers" at work: Very few known polynomial-time algorithms for natural problems have exponents above 3 or 4 (even though at discovery, the initial exponent may have been 30 or 40). However, many important natural problems that so far resist any efficient algorithms cannot at present be solved faster than in *exponential* time (which of course is totally impractical, even for small input data). This exponential gap gives great motivation for the definition of $\mathcal{P}$; reducing the complexity of such problems from exponential to (any) polynomial will be a huge conceptual improvement, likely involving new techniques.

3.2.2 Why worst case?

Another criticism of the definition of the class $\mathcal{P}$ is that a problem is deemed efficiently solvable if *every* input of length n can be solved in poly(n)-time. From a practical standpoint, it would suffice that the instances we care about (e.g., those generated by our application, be it in industry or nature) be solved quickly by our algorithms, and the rest can take a long time. Perhaps it suffices that "typical" instances be solved quickly. Of course, understanding what instances arise in practice is a great problem in itself, and a variety of models of typical behavior and algorithms for them are studied (and we shall touch upon this in section 4.4). The clear advantage of worst-case analysis is that we don't have to worry about which instances arise—they will all be solved quickly by what we call an "efficient algorithm". This notion "composes" well (i.e., when one algorithm is using another). Moreover, it accounts for adversarial situations where an input (or more generally, external behavior) is generated by an unknown opponent, who wishes to slow down the algorithm (or system). Modeling such adversaries is crucial in such fields as cryptography and error correction, and it is facilitated by worst-case analysis. Finally, as mentioned, this notion turned out to reveal a very elegant structure of the complexity universe, which inspired the more refined study of average-case and instance-specific theories.

Understanding the class $\mathcal{P}$ is central. Numerous computational problems arise (in theory and practice) that demand efficient solutions. Many algorithmic techniques were developed in the past four decades and enable solving many of these problems (see, e.g., the textbooks [CLR01, KT06]). These techniques drive the ultrafast home computer applications we now take for granted, like web searching, spell checking, data processing, computer game graphics, and fast arithmetic, as well as heavier duty programs used across industry, business, math, and science. But many more problems (some of which we shall meet soon), perhaps of higher

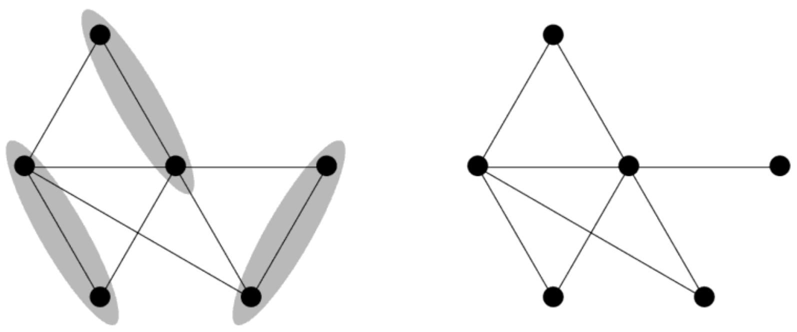


Figure 3. A graph with a perfect matching (left; matching is shown) and one without a perfect matching (right).

practical and theoretical value, remain elusive. The challenge of *characterizing* this fundamental mathematical object—the class $\mathcal{P}$ of efficiently solvable problems—is far beyond us at this point.

We end this section with some examples of nontrivial problems in $\mathcal{P}$ of mathematical significance from diverse areas. In each, the interplay between mathematical and computational understanding needed for the development of these algorithms is evident. Most examples are elementary in nature, but if some mathematical notion is unfamiliar, feel free to ignore that example (or possibly better, look up its meaning).

3.2.3 Some problems in $\mathcal{P}$

- **Perfect matching.** Given a graph, test whether it has a *perfect matching*, namely, a pairing of all its vertices such that every pair is an edge of the graph (see Figure 3). The ingenious algorithm of Edmonds [Edm65b] is probably the first nontrivial algorithm in $\mathcal{P}$, and as mentioned above, this paper was central to highlighting $\mathcal{P}$ as an important class to study. The structure of matchings in graphs is one of the most well-studied subjects in combinatorics (see, e.g., [LP09]).

- **Primality testing.** Given an integer, determine whether it is prime.[3] Gauss literally appealed to the mathematical community to find an efficient algorithm, but it took two centuries to resolve. The story of this recent achievement of Agrawal, Kayal, and Saxena [AKS04] and its history is beautifully recounted by Granville in [Gra05]. Of course, there is no need to elaborate on how central prime numbers are in mathematics (and even in popular culture).

- **Planarity testing.** Given a graph, determine whether it is *planar*. Namely, can it be embedded in the plane without any edges crossing? (Try to determine this for the graphs in Figure 3 and those in Figure 5.) A sequence of *linear* time algorithms for this basic problem was discovered, starting with the paper of Hopcroft and Tarjan [HT74].

[3]For example, try to determine the answer for $X - 1$ and $X + 1$, where $X = 6797727 \times 2^{15328}$.

use polynomials to define both terms. A candidate proof y for the claim $x \in C$ must have length at most polynomial in the length of x. And the verification that a given y indeed proves the claim $x \in C$ must be checkable in polynomial time (via a verification algorithm we will call V_C). Finally, if $x \notin C$, no such y should exist. Let us formalize this definition.

Definition 3.3 (The class $\mathcal{NP}$)**.** The set C is in the class $\mathcal{NP}$ if there is a function $V_C \in \mathcal{P}$ and a constant k such that

- If $x \in C$, then $\exists y$ with $|y| \leq k \cdot |x|^k$ and $V_C(x, y) = 1$;
- If $x \notin C$, then $\forall y$ we have $V_C(x, y) = 0$.

From a logic standpoint, each set C in $\mathcal{NP}$ may be viewed as a set of theorems in the complete and sound proof system defined by the *verification process* V_C.

A sequence y that "convinces" V_C that $x \in C$ is often called a *witness* or *certificate* for the membership of x in C. Again, we stress that the definition of $\mathcal{NP}$ is not concerned with how difficult it is to come up with a witness y, but rather only with the efficient verification using y that $x \in C$. The witness y (if it exists) can be viewed as given by an omnipotent entity, or simply guessed. Indeed, the acronym $\mathcal{NP}$ stands for "Nondeterministic Polynomial time," where the nondeterminism captures the ability of a *hypothetical* nondeterministic machine to "guess" a witness y (if one exists) and then verify it deterministically.

Nonetheless, the complexity of finding a witness is, of course, important, as it captures the *search problem* associated with $\mathcal{NP}$ sets. Every decision problem C (indeed, every verifier V_C for C) in $\mathcal{NP}$ defines a natural search problem associated with it: Given $x \in C$, *find* a short witness y that "convinces" V_C of this fact. A correct solution to this search problem can be efficiently verified by V_C, by definition.

It is clear that finding a witness (if one exists) can be done by brute-force search: as witnesses are short (of length poly(n) for a length-n input), one can enumerate all possible ones, and to each apply the verification procedure. However, this enumeration takes *exponential time* in n. The major question of this chapter (and this book, and the theory of computation!) is whether much faster algorithms than brute-force exist for *all* $\mathcal{NP}$ problems.

While it is usually the search problems that arise more naturally, it is often more convenient to study the decision versions of these problems (namely, whether a short witness exists). In almost all cases, both decision and search versions are computationally equivalent.[9]

Here is a list of some problems (or rather properties) in $\mathcal{NP}$, besides *THEOREMS* and *COMPOSITES* which we saw above. Note that some are variants of the problems in the similar list we gave for the class $\mathcal{P}$. However, we have no idea whether any of these are in $\mathcal{P}$. It is a good exercise (easy for most but not all examples) for the reader to define for each of them the short, easily verifiable witnesses for inputs having the property.[10]

Some problems in $\mathcal{NP}$:

[9]A notable possible exception is the set *COMPOSITES* and the suggested verification procedure for it, accepting as witness a nontrivial factor. Note that while *COMPOSITES* $\in \mathcal{P}$ is a decision problem, the related search problem is equivalent to integer factorization, which is not known to have an efficient algorithm.

[10]The one difficult exception is Matrix Group Membership, which, if you cannot resolve yourself, peek in the beautiful [BS84].

- **Hamiltonian cycles in graphs.** The set of graphs having a Hamilton cycle, namely, a cycle of edges passing through every vertex exactly once (see Figure 5).[11]
- **Factoring integers.** Triples of integers (x, a, b), such that x has a prime factor in the interval $[a, b]$.
- **Integer programming.** Sets of linear inequalities in many variables that have an integer solution.
- **Matrix group membership.** Triples (A, B, C) of invertible matrices (say, over $\mathbb{F}_2$) of the same size, such that A is in the subgroup generated by B, C.
- **Graph isomorphism.** Pairs of graphs that are isomorphic, namely, having a bijection between their vertex sets that extends to a bijection on their edge sets. (Find which pairs of graphs in Figure 5 are isomorphic.)
- **Polynomial root**. Multivariate polynomials of degree 3 over $\mathbb{F}_2$ that have a root (namely, an assignment to the variables on which it evaluates to 0).

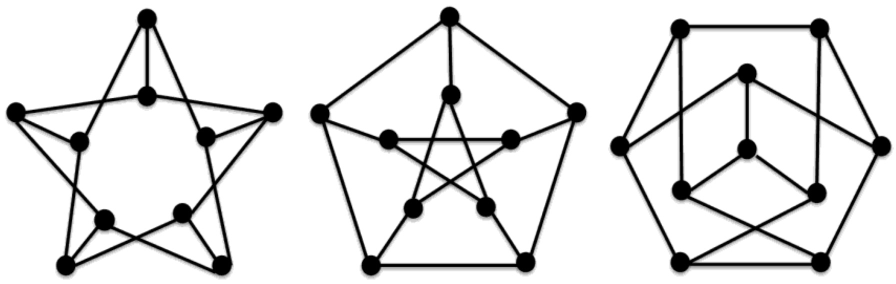


Figure 5. Which of these graphs are Hamiltonian? Which pairs of these graphs are isomorphic?

It is evident that decision problems in $\mathcal{P}$ are also in $\mathcal{NP}$. The verifier V_C is simply taken to be the efficient algorithm for C, and the witness y can be the empty sequence.

Corollary 3.4. $\mathcal{P} \subseteq \mathcal{NP}$.

But can we solve all $\mathcal{NP}$ problems efficiently? Can we vastly improve the trivial "brute-force" exponential time algorithm mentioned above to polynomial time for all $\mathcal{NP}$ problems? This is the celebrated $\mathcal{P}$ vs. $\mathcal{NP}$ question.

Open Problem 3.5. Is $\mathcal{P} = \mathcal{NP}$?

The definition of $\mathcal{NP}$, and the explicit $\mathcal{P} = \mathcal{NP}$? question (and much more that we will soon learn about) appeared formally first (independently and in slightly

[11]This problem is a special case of the well-known "Traveling Salesman Problem" (TSP), seeking the shortest such tour in a graph with edge lengths given.

different forms) in the papers of Cook [Coo71] and Levin [Lev73] in the early 1970s, one researcher in America and the other in the Soviet Union. However, both the definition and question appeared informally earlier, again independently in the East and West, but with similar motivations. They all struggle with the *tractability* of solving problems for which *finite* algorithms exist, including finding finite proofs of theorems, short logical circuits for Boolean functions, isomorphism of graphs, and a variety of optimization problems of practical and theoretical interest. In all these examples, exhaustive search was an obvious but exponentially expensive solution, and the goal of improving it by a possibly more clever (and faster) algorithm was sought, hopefully one of polynomial complexity (namely, in $\mathcal{P}$). The key recognition was identifying the superclass $\mathcal{NP}$ that so neatly encompasses almost all the seemingly intractable problems mentioned above that people really cared about and struggled with.

The excellent survey of Sipser [Sip92] describes this history and gives excerpts from important original papers. Here I mention only a few precursors to the papers above. In the Soviet Union, Yablonskii and his school studied *Perebor*, literally meaning "exhaustive, brute-force search," and Levin's paper continues this line of research (see Trakhtenbrot's survey [Tra84] of this work, including a corrected translation of Levin's paper). In the West, Edmonds [Edm66] was the first to explicitly suggest "good characterization" of the short, efficiently verifiable type (which he motivates by a teacher-student interaction, although in a slightly stricter sense than $\mathcal{NP}$, which we will soon meet in Section 3.5). But already in 1956, a decade earlier, a remarkable letter (discovered only in the 1990s, see original and translation in [Sip92]) written by Gödel to von Neumann essentially introduces $\mathcal{P}$, $\mathcal{NP}$, and the $\mathcal{P}$ vs. $\mathcal{NP}$ question in rather modern language (see Section 1.2 of [Wig10b] for more). In particular, Gödel raises this fundamental problem of overcoming brute-force search, exemplifies that it is sometimes nontrivially possible, and demonstrates clearly how aware he was of the significance of this problem. Unfortunately, von Neumann was already dying of cancer at the time, and it is not known whether he ever responded or Gödel had further thoughts on the subject. It is interesting that these early papers have different expectations as to the resolution of the $\mathcal{P}$ vs. $\mathcal{NP}$ question (in the language of their time): Gödel speculates that *THEOREMS* might be in $\mathcal{P}$, while Edmonds [Edm67a] conjectures that the Traveling Salesman problem is not in $\mathcal{P}$.

A very appealing feature of the $\mathcal{P}$ vs. $\mathcal{NP}$ question (which was a source of early optimism about its possible quick resolution) is that it can be naturally viewed as a *bounded* analog of the decidability question from computability theory, which we already discussed implicitly in Chapter 2. To see this, replace the *polynomial-time* bound by a *finite* bound in both classes. For $\mathcal{P}$, the analog becomes all problems having finite algorithms, namely the decidable problems, sometimes called *Recursive* problems and denoted by $\mathcal{R}$. For $\mathcal{NP}$, the analog is the class of properties for which membership can be certified by a finite witness via a finite verification algorithm. This class of problems is called *Recursively Enumerable*, or $\mathcal{RE}$. It is easy to see that most problems mentioned in Chapter 1 are in this class. For example, consider the properties defined by problems 1 and 4 from Chapter 2, respectively the solvable Diophantine equations and the theorems provable in Peano arithmetic. In the first, an integer root of a polynomial is clearly a finite witness that can be easily verified by evaluation in finite time. In the second, a Peano proof of a given theorem is a finite witness, and the chain of deductions of the proof can be easily verified in finite time. Thus, both problems are in $\mathcal{RE}$. We already know that both problems are undecidable (namely, are not in $\mathcal{R}$) and so can conclude that $\mathcal{R} \neq \mathcal{RE}$.

With nearly a half century of experience, we realize that resolving $\mathcal{P}$ vs. $\mathcal{NP}$ is much harder than $\mathcal{R}$ vs. $\mathcal{RE}$. A possible analogy (with a much longer history) is the difficulty of resolving the Riemann Hypothesis, though we have known for millennia that there are infinitely many primes. In both contexts, what we already know is very qualitative, separating the finite and infinite, and what we want to know are very precise, quantitative versions. Also for both problems, some weak quantitative results were proved along the way. The prime number theorem is a much finer quantitative result about the distribution of primes than their infinitude. In this book, we will discuss analogous quantitative progress on the computational complexity of natural problems. In both cases, the long-term goals seem to require much deeper understanding of the respective fields and far better tools and techniques. Incidentally, we will discuss a completely different analogy between the $\mathcal{P}$ vs. $\mathcal{NP}$ question and the Riemann hypothesis in Chapter 8.

3.4 The $\mathcal{P}$ vs. $\mathcal{NP}$ question: Its meaning and importance

Should you care about the $\mathcal{P}$ vs. $\mathcal{NP}$ question? The previous sections make a clear case that it is a very important question of computer science. It is also a precise mathematical question. How about its importance for mathematics? For some mathematicians, the presence of this question in the list of seven Clay Millennium Prize Problems [CJW06], alongside the Riemann Hypothesis and the Poincaré conjecture (which has since been resolved), may be sufficient reason to care. After all, these problems were selected by top mathematicians in the year 2000 as major challenges for the next millennium, each carrying a prize of one million dollars for its solution.

In this section, I hope to explain the ways in which the $\mathcal{P} = \mathcal{NP}$ question is unique not only among the Clay problems but also among all mathematics questions ever asked, in its immense practical and scientific importance, and its deep philosophical content. In a (very informal, sensational) nutshell, it can be summarized as follows:

Can we solve all the problems we can "legitimately" hope to solve?

where the royal "we" can stand for anyone or everyone, representing the general human quest for knowledge and understanding. In particular, this phrasing of the $\mathcal{P} = \mathcal{NP}$ question clearly addresses the possibility of resolving extant and future conjectures and open problems raised by mathematicians (at the very least, problems regarding classifications of mathematical objects).

To support this overarching interpretation of the $\mathcal{P} = \mathcal{NP}$ question, let us try to understand at a high level and in intuitive terms which problems occupy these two important classes. In fact, we have already intuitively identified the class $\mathcal{P}$ with a good approximation of all problems we can solve (efficiently, e.g., in our lifetimes). So next we embark on intuitively identifying $\mathcal{NP}$ as a good approximation of all "interesting" problems: those we are really investing effort in trying to solve, believing that we possibly can. Note that any argument for this interpretation will have to explain why undecidable problems (that are clearly not in $\mathcal{P}$) are not really "interesting" in this sense.

The very idea that all (or even most, or even very many) "interesting" problems can be mathematically identified is certainly audacious. Let us consider it, progressing slowly. We caution that this discussion is mainly of a philosophical nature, and the arguments I make here are imprecise and informal, representing my personal views. I encourage the reader to poke holes in these arguments, but I also challenge you to consider whether counterexamples found to general claims made here are typical or exceptional. After this section, we shall soon return to the sure footing of mathematics!

So, which problems occupy $\mathcal{NP}$? The class $\mathcal{NP}$ turns out to be extremely rich. There are literally thousands of $\mathcal{NP}$ problems in mathematics, optimization, artificial intelligence, biology, physics, economics, industry, and more that arise naturally out of very different necessities, and whose efficient solutions will benefit us in numerous ways. *What is common to all these possibly hard problems, which nevertheless separates them from certainly hard problems (like undecidable ones)?*

To explore this, it is worthwhile to consider a related question: *What explains the abundance of so many natural, important, diverse problems in the class $\mathcal{NP}$?* After all, this class was defined as a technical, mathematical notion by computational theorists. Probing the intuitive meaning of the definition of $\mathcal{NP}$, we will see that it captures many tasks of human endeavor *for which a successful completion can be easily recognized.* Consider the following professions, and the typical tasks they are facing (this list will be extremely superficial, but nevertheless instructive):

- **Mathematician:** Given a mathematical claim, come up with a proof for it.
- **Scientist:** Given a collection of data on some phenomena, find a theory explaining it.
- **Engineer:** Given a set of constraints (on cost, physical laws, etc.), come up with a design (of an engine, bridge, laptop, etc.) that meets these constraints.
- **Detective:** Given the crime scene, find "who done it."

Consider what may be a common feature of this multitude of tasks. I claim that in almost all cases, "we can tell" a good[12] solution when we see one (or we at least believe that we can). Simply put, *would you embark on a discovery process if you didn't expect to recognize what you set out to find?* It would be good for the reader to consider this statement seriously and try to look for counterexamples. Of course, in different settings the "we" above may refer to members of the academic community, consumers of various products, or the juries in different trials. I have had many fun discussions, especially after lectures on the subject, of whether scientists or even artists are indeed in the mental state described. I believe they are. It seems to me that in these cases, the very decision to expose (or not) to others of our creations typically follows the application of such a "goodness test" to our work. Thus, embarking on any such task we undertake, we (implicitly or explicitly) expect the solution (or creation) we come up with to essentially bear the burden of proof of goodness that we can test; namely, be *short* and *efficiently verifiable*, just as in the definition of $\mathcal{NP}$.

[12]In this context, "good" may mean "optimal," or "better than previous ones," or "publishable," or any criterion we establish for ourselves.

3.5 The class co$\mathcal{NP}$, the $\mathcal{NP}$ vs. co$\mathcal{NP}$ question, and efficiently characterizable structures

We have discussed efficient computation and efficient verification. Now let us turn to define and discuss efficient *characterization* of properties. Note that attempts, mainly in combinatorics, graph theory, and optimization, to find "good" characterizations (some successful ones, as for perfect matchings and Euler tours in graphs, and some failed ones, as for Hamiltonian cycles and colorings in graphs), were central to elucidating the definitions and importance of the concepts and classes in this chapter. Many of these, and the focus on formally defining the notion of "good" (in characterizations as well as in algorithms), go back Edmonds' early optimization papers, mainly [Edm66].

Fix a property $C \subseteq \mathbf{I}$. We already have the interpretations

- $C \in \mathcal{P}$ if it is easy to compute if an object x has property C,
- $C \in \mathcal{NP}$ if it is easy to certify that an object x has property C,

to which we now add

- $C \in \text{co}\mathcal{NP}$ if it is easy to certify that an object x *does not have* property C,

where we formally define the class as follows.

Definition 3.7 (The class co$\mathcal{NP}$)**.** A set C is in the class co$\mathcal{NP}$ iff its complement $\bar{C} = \mathbf{I} \setminus C$ is in $\mathcal{NP}$.

For example, the set *PRIMES* of all prime numbers is in co$\mathcal{NP}$, since its complement *COMPOSITES* is in $\mathcal{NP}$. Similarly, the set of non-Hamiltonian graphs is in co$\mathcal{NP}$, since its complement, the set of all Hamiltonian graphs, is in $\mathcal{NP}$.

While the definition of the class $\mathcal{P}$ is symmetric,[14] the definition of the class $\mathcal{NP}$ is *asymmetric*. Having nice certificates that a given object has property C by no means automatically entails nice certificates that a given object does *not* have this property.

Indeed, when we can do both, namely, having nice certificates for both the set and its complement, we are achieving one of mathematics' holy grails of understanding structure, namely, *necessary and sufficient* conditions, sometimes phrased as a *characterization* or a *duality theorem*. As we well know, such characterizations are rare. When insisting (as I shall) that the certificates are furthermore *short, efficiently verifiable* ones,[15] such characterizations are even rarer. This leads to the following conjecture.

Conjecture 3.8. $\mathcal{NP} \neq \text{co}\mathcal{NP}$.

Note that this conjecture implies $\mathcal{P} \neq \mathcal{NP}$. We shall discuss at length refinements of this conjecture in Chapter 6 on proof complexity.

Despite the shortage of such *efficient* characterizations (namely, properties that are simultaneously in $\mathcal{NP} \cap \text{co}\mathcal{NP}$), they nontrivially exist. This class was introduced by Edmonds [Edm66], who called them problems with *good* characterization.

[14]Having a fast algorithm to determine whether an object has a property C is equivalent to having a fast algorithm for the complementary set $\bar{C}$. In other words, $\mathcal{P} = \text{co}\mathcal{P}$.

[15]There are many famous duality theorems in mathematics that do not conform to this strict efficiency criterion (e.g., Hilbert's Nullstellensatz).

Here is a list of some exemplary ones, following important theorems of (respectively) Menger, Dilworth, Farkas, von Neumann, and Pratt. I informally explain the $\mathcal{NP}$ and co$\mathcal{NP}$ witnesses for most, which can be seen to be efficiently verifiable. Of course, the crux is that for each of these problems, *every* instance of the problem possesses one such witness: having the property or violating it.

Efficient duality theorems: problems in $\mathcal{NP} \cap$ co$\mathcal{NP}$

- **Graph k-connectivity.** The set of graphs in which *every* pair of vertices is connected by (a given number) k disjoint paths. Here the NP-witness is a collection of such k paths between every pair, and the co$\mathcal{NP}$-witness is a *cut* of $k-1$ vertices whose removal disconnects some pair in the graph.

- **Partial order width.** Finite partially ordered set (poset) whose largest *antichain* (a set of pairwise incomparable elements) has at least (a given number) w elements. Here the $\mathcal{NP}$-witness is an antichain of w elements, and the co$\mathcal{NP}$-witness is a partition of the given poset to $w-1$ *chains* (totally ordered sets).

- **Linear programming.** Systems of consistent linear inequalities. Here an $\mathcal{NP}$-witness is a point satisfying all inequalities. A co$\mathcal{NP}$-witness is a linear combination of the inequalities producing the contradiction $0 > 1$.[16]

- **Zero-sum games.**[17] Finite zero-sum games (described by a real payoff matrix) in which the first player can gain at least v (some given value). Here the $\mathcal{NP}$-witness is a strategy for the first player (namely, a probability distribution on the rows), which guarantees her a payoff of v, and the co$\mathcal{NP}$-witness is a strategy for the second player (namely, probability distribution on the columns), which guarantees that he pays less than v.

- **Primes.** Prime numbers. Here the co$\mathcal{NP}$-witness is simple: two nontrivial factors of the input. The reader is encouraged to attempt to find the $\mathcal{NP}$-witness: a short certificate of primality. It requires only very elementary number theory.[18]

The known relations of $\mathcal{P}$, $\mathcal{NP}$, and co$\mathcal{NP}$ are depicted in Figure 6. The examples above, of problems in $\mathcal{NP} \cap$ co$\mathcal{NP}$ were chosen to make a point. At the time of their discovery, interest was seemingly focused only on characterizing these structures; it is not clear whether efficient algorithms for these problems were sought as well. However, with time, all these problems turned out to be in $\mathcal{P}$, and their resolutions entered the Hall of Fame of efficient algorithms. Famous examples are the ellipsoid method of Khachian [Kha79] and the interior-point method of Karmarkar [Kar84], both for Linear Programming, and the breakthrough algorithm of Agrawal, Kayal, and Saxena [AKS04] for Primes.[19]

[16]This duality generalizes to other convex bodies given by more general constraints, like *semi-definite* programming. Such extensions include the Kuhn-Tucker conditions and the Hahn-Banach theorem.

[17]This problem was later discovered to be equivalent to linear programming.

[18]Hint: Roughly, the witness consists of a generator of Z_p^*, a factorization of $p-1$, and a recursive certificate of the same type for each of the factors.

[19]It is interesting that assuming the Generalized Riemann Hypothesis, a simple polynomial-time algorithm was given 30 years earlier by Miller [Mil76].

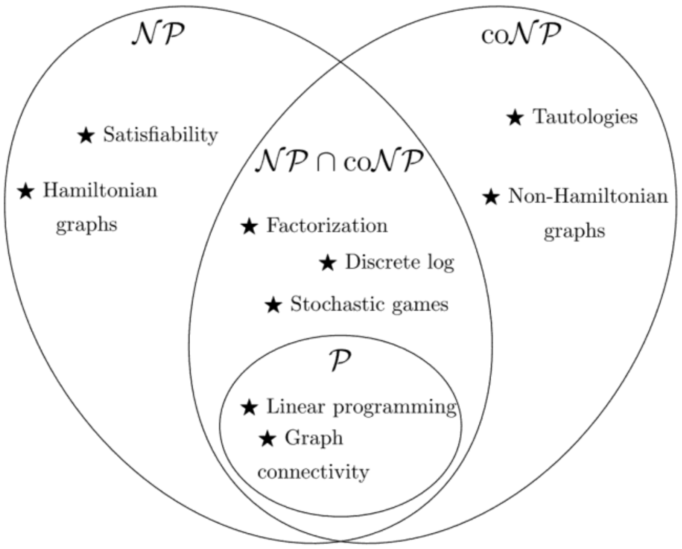


Figure 6. $\mathcal{P}$, $\mathcal{NP}$, and co$\mathcal{NP}$.

Is there a moral to this story? Only that sometimes, when we have an efficient characterization of structure, we can hope for more: efficient algorithms. Indeed, a natural stepping stone toward an elusive efficient algorithm may first be to get an efficient characterization.

Can we expect this magic to always happen? Is $\mathcal{NP} \cap \text{co}\mathcal{NP} = \mathcal{P}$? We do not have too many examples of problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ that have resisted efficient algorithms. Some of the famous, like integer factoring and discrete logarithms,[20] arise from *one-way* functions which underlie cryptography (we discuss these in Section 4.5). Note that while they are not known to be hard, humanity literally banks on their intractability for electronic commerce security. Yet another famous example, for which membership in $\mathcal{NP}$ and in co$\mathcal{NP}$ are highly nontrivial (respectively proved in [Lac15] and [HLP99]) is the *unknottedness* problem, namely, testing whether a knot diagram represents the trivial knot. A very different example is Shapley's *stochastic games*, studied by Condon in [Con92], for which no efficient algorithm is known. However, we have seen that many problems first proved to be in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ eventually were found to be in $\mathcal{P}$. It is hard to generalize from so few examples, but the general belief is that the two classes are different.

Conjecture 3.9. $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$.

Note that this conjecture implies $\mathcal{P} \neq \mathcal{NP}$.

We now return to developing the main mechanism, which will help us study such questions: *efficient reductions* and *completeness*.

[20]Which have to be properly defined as decision problems.

3.6 Reductions: A partial order of computational difficulty

In this section, we deal with relating the computational difficulty of problems for which we have no efficient solutions (yet).

Recall that we can regard any classification problem (on finitely described objects) as a subset of our set of inputs $\mathbf{I}$. Efficient reductions provide a natural partial order on such problems that captures their relative difficulty. Note that reductions are a primary tool in computability and recursion theory, from which computational complexity developed. There, reductions were typically simply *computable* functions, whereas the focus of computational complexity is on *efficiently computable* ones. While we concentrate here on time efficiency, the field studies a great variety of other resources; limiting these in reductions is as fruitful as limiting them in algorithms. The following crucial definition is depicted in Figure 7.

Definition 3.10 (Efficient reductions)**.** Let $C, D \subset \mathbf{I}$ be two classification problems. $f\colon \mathbf{I} \to \mathbf{I}$ is an efficient reduction from C to D if $f \in \mathcal{P}$ and for every $x \in \mathbf{I}$ we have $x \in C$ iff $f(x) \in D$. In this case, we call f an *efficient reduction* from C to D. We write $C \leq D$ if there is an efficient reduction from C to D.

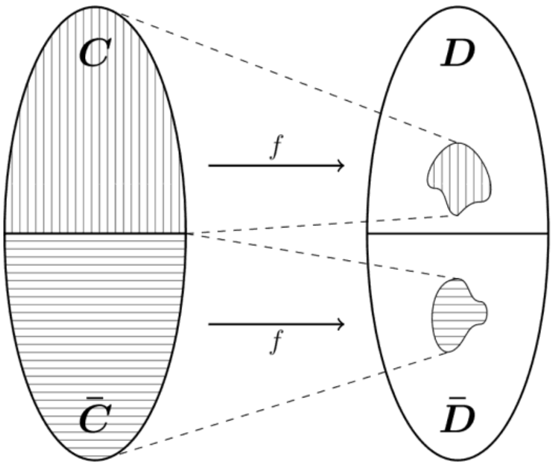


Figure 7. A schematic illustration of a reduction between two classification problems.

So, if $C \leq D$, then solving the classification problem C is computationally not much harder than solving D (up to polynomial factors in the running time).[21] If we have both $C \leq D$ and $D \leq C$ then C and D are computationally equivalent (again, up to polynomial factors). This gives formal meaning to the word “equivalent” in Theorem 3.1.

The definition of efficient computation (more precisely, that the composition of two polynomials is a polynomial) allows two immediate but important observations on the usefulness of efficient reductions. Please verify both for yourself! First, that

[21] In particular, if $C \in \mathcal{P}$ then it is not much harder than trivial problems D (e.g. D might ask to distinguish sequences starting with 0 from those starting with 1). The reduction in this case would simply solve C.

indeed $\leq$ is *transitive*, and thus defines a partial order on classification problems. Second, one can compose (as in Figure 8) an efficient algorithm for one problem and an efficient reduction from a second problem to get an efficient algorithm for the second. Specifically, if $C \leq D$ and $D \in \mathcal{P}$, then also $C \in \mathcal{P}$.

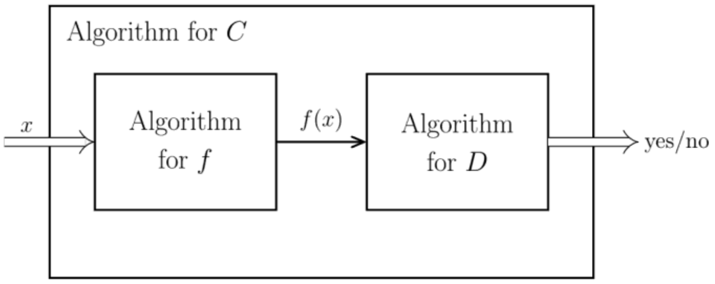


Figure 8. Composing a reduction and an algorithm to create a new algorithm.

As noted, $C \leq D$ means that solving the classification problem C is computationally not much harder than solving D. In some cases, one can replace "computationally" by the (vague) term "mathematically." To be useful mathematically and allow better understanding of one problem in terms of another may require more properties of the reduction f than merely being efficiently computable. For example, we may want f to be a linear transformation, or a low-degree polynomial map, and indeed in some cases (as we will see e.g. in Chapter 12) this is possible. When such a connection between two classification problems (which look unrelated) can be proved, it can mean the importability of techniques from one area to another.

The power of efficient reductions to relate seemingly unrelated notions will unfold in later sections. We shall see that they can relate not only classification problems but also such diverse concepts as hardness to randomness; average-case to worst-case difficulty; proof length to computation time; and last but not least, the security of electronic transactions to the difficulty of factoring integers. In a sense, *efficient reductions are the backbone of computational complexity.* Indeed, given that polynomial time reductions can do all these wonders, no wonder we have a hard time characterizing the class $\mathcal{P}$!

3.7 Completeness: Problems capturing complexity classes

We now return to classification problems. The partial order of their difficulty, provided by efficient reductions, allows us to define the *hardest* problems in a given class of problems. Let $\mathcal{C}$ be any collection of classification problems (namely, every element of $\mathcal{C}$ is a subset C of $\mathbf{I}$). In this chapter we are mainly concerned about the class $\mathcal{C} = \mathcal{NP}$. But later in the book we will see this important idea recur for other *complexity classes* (namely classes of problems defined by the resources required to solve them, like $\mathcal{NP}$) .

Definition 3.11 (Hardness and completeness)**.** A problem D is called $\mathcal{C}$*-hard* if for every $C \in \mathcal{C}$, we have $C \leq D$. If we further have that if $D \in \mathcal{C}$, then D is called $\mathcal{C}$*-complete.*

In other words, if D is $\mathcal{C}$-complete, it is a hardest problem in the class $\mathcal{C}$: if we manage to solve D efficiently, we have done so automatically for all other problems

$By + C = 0$, then in fact there is a short one,[24] indeed a solution whose length (in bits) is linear in the lengths of A, B, C. Thus, a short witness is simply a root (x, y). But $KNOT$ is an exception, and the short witnesses for the knot having a small genus requires Haken's algorithmic theory of normal surfaces, considerably enhanced (even short certificates for unknottedness in $\mathbb{R}^3$ are hard to obtain; see [HLP99]). Let us discuss what these $\mathcal{NP}$-completeness results mean, first about the relationship between the three sets, and then about each individually.

The proofs that these problems are complete follow by reductions from (variants of) *SAT*. The discrete, combinatorial nature of these reductions may cast doubt on the possibility that the computational equivalence of these problems implies the ability of real "technology transfer" between, for example, topology and number theory. Nevertheless, now that we know of the equivalence, perhaps simpler and more direct reductions can be found between these problems. Moreover, we stress again that reductions translate between witnesses as well. Namely, for any instance, say $(M, K, G) \in KNOT$, if we translate it using this reduction to an instance $(A, B, C) \in 2DIO$ and happen (either by sheer luck or special structure of that equation) to find an integer root, the same reduction will translate that root back to a description of a genus G manifold that bounds the knot K. Today many such $\mathcal{NP}$-complete problems are known throughout mathematics, and for some pairs, the equivalence can be mathematically meaningful and useful (as it is between some pairs, of computational problems).

Let us discuss the simplest of the three reductions above, namely, from SAT to $3COL$. If you have never seen one, it should be a mystery: The two problems talk about different worlds, one of logic and the other of graph theory. Both are difficult problems, but the reduction should be easy (namely, efficiently computable). The key to this reduction, as well as to almost any other, is the locality of computation! This of course is evident in SAT; a formula is composed from Boolean gates, each of which performs a simple, local operation. However, $3COL$ feels like a more global property.[25] The idea of this reduction is to focus on the individual gates of the input formula. We'll find a reduction that works for each gate and will compose the small ("gadget") graphs produced, mimicking the structure prescribed by the input formula. Let's elaborate this idea.

Here is how to transform the satisfiability problem for the (trivial, 1-gate) formula $x \vee y$ to a graph 3-coloring problem. We will actually transform the equation $x \vee y = z$ to a graph 3-coloring statement using the gadget graph shown in Figure 9. Check that it satisfies the following condition: In *every* legal 3-coloring of the graph with colors $\{0, 1, 2\}$, the colors of the vertices labeled x, y, z will be from $\{0, 1\}$, which will satisfy the equation $x \vee y = z$. One can easily construct such gadgets for the gates $\wedge$, $\neg$ as well. Now, to complete the reduction, the algorithm proceeds as follows. Given an arbitrary formula as input, it names its wires, builds a gadget graph for every gate, and identifies appropriate vertices in these to generate an output graph. By construction, it is 3-colorable if and only if the given formula was satisfiable. This is essentially the reduction in [Kar72], but we are not done yet: The gadget graph above is not planar (and hence the output graphs are also not planar). However, Stockmeyer [Sto73] gives another gadget that can eliminate

[24]Hint: If (x, y) is a root, so is $(x + B, y - A(2x + B))$.

[25]Consider, for example, a cycle on n vertices, where n is odd; it requires 3 colors, but if we remove any edge, it can be 2-colored.

crossings in planar embeddings of graphs without changing their 3-colorability. The reader is encouraged to find such a gadget. With this, the proof of Theorem 3.17 is complete.

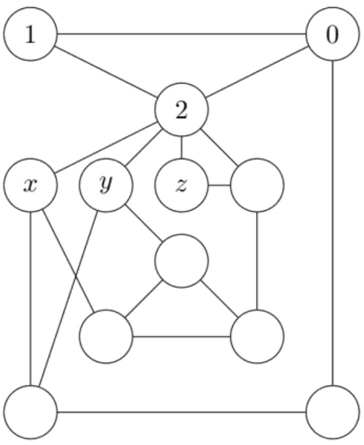


Figure 9. The gadget underlying the reduction from SAT to $3COL$.

We now list a few more $\mathcal{NP}$-complete problems of a different nature, to give a feeling for the breadth of this phenomenon. Some appear already in Karp's original article [Kar72]. Again, hundreds more can be found in Garey and Johnson's book, [GJ79], and by now, many thousands are known:

- **Hamiltonian cycle.** Given a graph, is there a simple cycle of edges going through every vertex precisely once?
- **Subset-sum.** Given a sequence of integers $a_1, \ldots, a_n$ and b, is there a subset J such that $\sum_{i \in J} a_i = b$?
- **Integer programming.** Given a polytope in $\mathbb{R}^n$ (by its bounding hyperplanes), does it contain an integer point?
- **Clique.** Given a graph and an integer k, are there k vertices with all pairs mutually adjacent?
- **Quadratic equations.** Given a system of multivariate polynomial equations of degree at most 2, over a finite field (say, $\mathbb{F}_2$), do they have a common root?
- **Shortest lattice vector.** Given a lattice L in $\mathbb{R}^n$ and an integer k, is the shortest nonzero vector of L of (Euclidean) length $\leq k$?

3.10 The nature and impact of $\mathcal{NP}$-completeness

$\mathcal{NP}$-completeness is a unique scientific discovery—there seems to be no precisely defined scientific notion that even comes close to being pervasive in so many fields of science and engineering! We start with its most immediate impact, in computer science itself, move to mathematics, and then to science and beyond. Some of this discussion will be become more meaningful (and impressive) as you read further through the book, and in detail in the last chapter. More can be found in, for example, Papadimitriou's retrospective on the subject [Pap97]. Curiously, that paper reports that electronic search (new at the time) revealed thousands of science and math papers with the phrase "$\mathcal{NP}$-complete" in them; today, more than 20 years later, this number is far larger!

As mentioned, starting with Karp's paper [Kar72], an explosion of $\mathcal{NP}$-completeness results followed quickly in every corner and subfield of computer science. This is easy to explain. Most individuals in the field of computer science, from academics to industry programmers, are busy seeking efficient algorithms for numerous computational problems. How can one justify failure to find such an algorithm? In the absence of any techniques for proving intractability, the next best thing was proving that the computational problem at hand was $\mathcal{NP}$-complete (or $\mathcal{NP}$-hard), which means that finding such an efficient algorithm for it would imply an efficient algorithm for numerous others, which many others failed to solve. In short, failing to prove $\mathcal{P} = \mathcal{NP}$ is a very powerful excuse, and $\mathcal{NP}$-completeness is an excellent stamp of hardness. Every professional of the field knows this! While $\mathcal{NP}$-completeness is a negative result (basically showing that what we want is impossible), such negative results had a positive impact. As problems do not go away when you declare them $\mathcal{NP}$-complete and still demand solutions, weaker solution concepts for them were developed. For example, for optimization problems, people attempted to find good approximation algorithms. Moreover, given that $\mathcal{NP}$-completeness only captures worst-case hardness, people developed algorithms that work well "on average" and heuristics that seem to work well on inputs that "show up in practice." This direction needs lots more theory, especially in light of the recent practical success of *deep networks*, that is far from understood. A variety of quality criteria and models were developed for different relaxations of efficient solvability, leading to analogous complexity theories, which enable researchers to argue hardness as well; some of these models will be discussed later in the book. A major such theory, able to argue $\mathcal{NP}$-completeness for approximation problems, and actually pinpoint in many cases the exact limits of efficient approximation, will be discussed in Section 10.3.

The next field to be impacted by $\mathcal{NP}$-completeness was mathematics. With some delay, $\mathcal{NP}$-completeness theorems started showing up in most mathematical disciplines, including algebra, analysis, geometry, topology, combinatorics, and number theory. This "intrusion" may seem surprising, as most questions that mathematicians ask themselves are not algorithmic. However, existence theorems for a variety of objects beg the question of having "explicit" descriptions of such objects. Moreover, in many fields, one actually needs to find such objects. Mathematics is full of a variety of constructions, done by hand long ago and by numerous libraries of computer programs that are essential for progress, and hence, their efficiency is also essential. Like computer scientists, mathematicians adopted the notion of a polynomial-time algorithm as a first cut at defining "efficient" and "explicit." Thus, description and construction problems that are $\mathcal{NP}$-complete were extremely useful to set limits on the hopes of achieving these properties. Furthermore, in mathematics, such $\mathcal{NP}$-completeness results signified an underlying "mathematical nastiness" of the structures under study. For example, as explained in Section 3.5, an efficient characterization of a property that is $\mathcal{NP}$-complete will imply that $\mathcal{NP} = \text{co}\mathcal{NP}$, and so it is unlikely (as we understand things today) that such a characterization exists. As in CS, so in math as well, such bad news begets good outcomes, nudging mathematicians into more productive directions: refining or specializing the properties under study, considering a variety of approximate notions, or simply being satisfied with sufficient and necessary conditions that are not complementary (as is needed for characterization).

The presence and impact on $\mathcal{NP}$-completeness in science is evidenced by the fact that such results (which are patently about computation) in biology, chemistry, economics, neuroscience, electrical engineering, and other fields are being proved not by computer scientists but by biologists, chemists, economists, neuroscientists, electrical engineers, and so forth. Moreover, these results are being published in the scientific journals of these very fields. And the numbers are staggering: A search for papers that contain the phrase "$\mathcal{NP}$-complete" or "$\mathcal{NP}$-completeness" prominently (in the title, abstract, or keywords) reveals that in *each* of these disciplines, there are *hundreds* of such papers, and many thousands more that mention these terms in the body of the paper. To obtain such results, these thousands of scientists needed to learn the concepts and proof methods of computational complexity, typically a foreign language to most (for instance, mathematical theorems rarely appear in science articles at all).

This phenomenon begs an explanation! Indeed, there are two questions to answer. What explains the abundance of $\mathcal{NP}$-completeness in these diverse disciplines? And why do their scientists bother making the unusual effort to prove these computational theorems?

One important observation is that scientists often study processes and try to build models that explain and predict them. Almost by definition, these are computational processes, namely, composed of a sequence of *simple, local steps*, like Turing machines, albeit manipulating not bits in computers but possibly neurons in the brain, proteins in the cell, atoms in matter, fish in a school, or stars in a galaxy. In other words, many models simply describe *algorithms* that nature uses for generating certain processes or behavior. A typical $\mathcal{NP}$-completeness result often refers to the limits of prediction by a particular model of some natural process. Here are some illustrative examples. In some existing models, it is $\mathcal{NP}$-complete to compute the following quantities: the minimal surface area that a given foam will settle into (in physics), the minimal energy configuration of a certain molecule (e.g., as in protein folding in biology), and the maximum social welfare of certain equilibria (in economics). Let's explore the meaning of such results to modeling natural phenomena.

We'll make two natural assumptions, which seem completely benign. First, that $\mathcal{P} \neq \mathcal{NP}$. Then $\mathcal{NP}$-completeness means that no efficient algorithm can compute the required quantities (e.g., in the examples just mentioned), at least for *some* instances. Second, that natural processes are inherently efficient algorithms, and so are the measurements we perform to extract these quantities at the end of the process. These two assumptions clash with each other, which seems to suggest at least one of two possible conclusions. One possibility is that the model (for which $\mathcal{NP}$-completeness was proved) is simply wrong (or incomplete) in describing reality. The other possibility is that the "hard" instances simply never occur in nature.[26] In both cases, $\mathcal{NP}$-completeness calls for a better understanding (e.g., refinement of the model at hand), a characterization of the instances for which the algorithm suggested by the model solves efficiently (and an argument that these conditions

[26]For example, it is quite possible that over billions of years of evolution, only proteins that are easily and efficiently foldable survived, and others became extinct.

are consistent with what we see in nature).

This idea has caused some researchers to propose that our underlying conjecture, $\mathcal{P} \neq \mathcal{NP}$, should be viewed as a *law of nature.* Perhaps the first explicit such occurrence is this quote,[27] from Volker Strassen's laudation [Str86] for Les Valiant on his Nevanlinna Prize: *"The evidence in favor of Cook's and Valiant's hypotheses is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures."* Note that at that time, the utility of this mathematical conjecture to science was not as well understood as it is today. How to precisely articulate this mathematical statement as a relevant law of nature is of course still interesting to debate. An intuitive desire is to let it play the same role that the second law of thermodynamics plays in science: Scientists would be extremely wary to propose a model that violates it. One suggestion, by Scott Aaronson, is a stronger statement about the real world: *There are no physical means to solve $\mathcal{NP}$-complete problems in polynomial time.* A host of possible physical means that were actually attempted is discussed in [Aar05].

There is still the mystery of the ubiquitous presence of $\mathcal{NP}$-completeness in practically every subfield of CS, math, and essentially all sciences. With hindsight, this is an amplified (and much more relevant) incarnation of the ubiquity of undecidability in all these disciplines. Both are explained by the fact that computation, viewed (as above) as any process evolving via a sequence of simple, local steps, is so ubiquitous. Similarly, descriptions of properties of systems with many parts (either desired properties or observed properties, which are typically the outcomes of such computations) are often given or modeled by sets of simple, local constraints on small subsystems of the whole. As it happens, for almost all choices of constraints, their mutual satisfaction for given instances is undecidable if the system is infinite and is $\mathcal{NP}$-complete if finite. In much rarer cases, they lead to (respectively) decidable or polynomial-time solvable problems. Understanding these phenomena, and delineating the types of constraints across the tractable/intractable barrier, is an active field of study and will be discussed further in Section 4.3.

Concluding this somewhat philosophical section, we note another major impact of $\mathcal{NP}$-completeness. Namely, that it served as a role model for numerous other notions of computational universality. $\mathcal{NP}$-completeness turned out to be an extremely flexible and extendible notion, allowing numerous variants, which enabled capturing universality in other (mainly computational, but not only) contexts. It led to the definitions of classes of problems solvable using very different resource bounds. In most cases, these classes were also shown to have complete problems, capturing the difficulty of the whole class under natural reductions, with the benefits described above (some examples will be discussed in Section 4.1 and then in later chapters). Much of the whole evolution of computational complexity, the theory of algorithms, and most other areas in theoretical computer science has been guided by the powerful approach of reduction and completeness. This progression has generated multiple theories of *intractability* in various settings and tools to understand and possibly curb or circumvent it (even though we are still mostly unable to prove intractability in most cases). The structures revealed by this powerful methodology send an important message to other disciplines.

[27]In this quote Cook's hypothesis is $\mathcal{P} \neq \mathcal{NP}$, and Valiant's hypothesis is what became known as $\mathcal{VP} \neq \mathcal{VNP}$ which we will discuss in Chapter 12.

One of the most interesting and fascinating is *MCSP*, the minimum circuit size problem[1] whose status is surveyed in [AH17].

- **Counting problems.** Fix an $\mathcal{NP}$ problem. Given an input, find the *number* of solutions (witnesses) for it. Many problems in enumerative combinatorics and in statistical physics fall into this category. Here too, a natural relaxation of counting problems is approximation: computing a number that is "close" to the actual count. The natural home of most of these problems is a class called $\#\mathcal{P}$. A most natural complete problem for this class is $\#SAT$, which asks to compute the number of satisfying assignments of a given formula (more generally, counting versions of typical $\mathcal{NP}$-complete classification problems are $\#\mathcal{P}$-complete). A remarkable complete problem for it is evaluating the *Permanent* polynomial,[2] or equivalently, counting the number of perfect matchings of a given bipartite graph. Thus, even counting versions of easy classification problems (e.g., testing if a perfect matching exists) can be $\#\mathcal{P}$-complete. This discovery, the definition of the class $\#\mathcal{P}$, and the complexity theoretic study of enumeration problems originates from Valiant's papers [Val79a, Val79c]. A surprising, fundamental result of Toda [Tod91] efficiently reduces quantified problems (above) to counting problems (in symbols, $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$).

- **Strategic problems.** Given a (complete information, 2-player) game, find an optimal strategy for a given player. Equivalently, given a position in the game, find the best move. Many problems in economics and decision theory, as well as playing well the game of chess, fall into this category. The natural home for most of these problems is the class $\mathcal{PSPACE}$ of problems solvable using a polynomial amount of memory (but possibly exponential time). Indeed, many such games (appropriately extended to families of games of arbitrary sizes, to allow asymptotics, and restricting the number of moves to be polynomial in "board size") become complete for $\mathcal{PSPACE}$. This characterization of the basic memory (or space) resource in computation in terms of alternation of quantifiers (namely, as game strategies) arises as well from [Sto76] and obviously extends the bounded alternation games described above (which defined $\mathcal{PH}$). A major, surprising understanding of polynomial space is the result $\mathcal{IP} = \mathcal{PSPACE}$ of [Sha92]. It establishes $\mathcal{PSPACE}$ as the home of all problems having efficient *interactive proofs* (an important extension of "written proofs" captured by $\mathcal{NP}$), as discussed in Section 10.1.

- **Total $\mathcal{NP}$ functions.** These are search problems seeking to find objects that are *guaranteed* to exist (like local optima, fixed points, Nash equilibria) and are certified by small witnesses. In many such problems, the input is an *implicitly defined*[3] exponentially large graph, (possibly weighted, possibly directed). The task is to find a vertex with some simple property, whose existence is guaranteed by a combinatorial principle. For example, that every directed acyclic graph has a sink (and so the task is to find one), or that every undirected graph has an even number of vertices of odd degree (and so

[1]The input to this problem is a Boolean circuit, and the problem is to determine if there exists a smaller circuit computing the same function. We will formally define circuits in Section 5.2.

[2]A sibling of the Determinant, which is discussed in Chapter 12.

[3]For example, via a program computing the neighbors of any given vertex.

the task is, given one such vertex, to find another). In the paper initiating this study, Papadimitriou [Pap94] defines several complexity classes, each captured by one such principle. These classes lie between (the search problems associated with) $\mathcal{P}$ and $\mathcal{NP}$. One important example is the class $\mathcal{PLS}$, for polynomial local search, in which a complete problem is finding a local minimum in a weighted directed graph. Another is the class $\mathcal{PPAD}$, for which a natural complete problem is (a discrete version of) computing a fixed point of a given function. Computing the Nash equilibrium in a given 2-player game is clearly in this class, as the proof of Nash's theorem (that every game has such an equilibrium) follows simply from Brouwer's fixed-point theorem. A major result [DGP09, CDT09] was proving the converse: establishing that finding a Nash equilibrium is a complete problem for this class. These classes of problems and their complexity were studied in [BCE$^+$95] through the framework of proof complexity, a subject we will discuss in Chapter 6.

Figure 10 shows some of the known inclusions between these classes and some problems in them. Note that even though SAT and $CLIQUE$ are $\mathcal{NP}$-complete, while *Perfect Matching* is in $\mathcal{P}$, their counting versions are all in $\#\mathcal{P}$, and indeed, all three are complete for this class (*Permanent* is the counting problem for perfect matchings).[4]

I shall not elaborate on these families of important problems and classes here. Some of them will be mentioned in subsequent sections, but I will not develop their complexity theory systematically. Note that the methodology of efficient reductions and completeness illuminates much of their computational complexity in the same way as for classification problems.

4.2 Between $\mathcal{P}$ and $\mathcal{NP}$

We have seen that $\mathcal{NP}$ contains a vast number of problems, but that in terms of difficulty, nearly all of those we have seen fall into one of two equivalence classes: $\mathcal{P}$, which are all efficiently solvable, and $\mathcal{NP}$-complete. Of course, if $\mathcal{P} = \mathcal{NP}$, the two classes are the same. But assuming $\mathcal{P} \neq \mathcal{NP}$, is there anything else? Ladner [Lad75] proved the following result.

Theorem 4.1 [Lad75]. *If $\mathcal{P} \neq \mathcal{NP}$, then there are infinitely many levels of difficulty in $\mathcal{NP}$. More precisely, there are sets $C_1, C_2, \ldots$ in $\mathcal{NP}$ such that for all i, we have $C_i \leq C_{i+1}$, but $C_{i+1} \not\leq C_i$.*

So, there is a lot of "dark matter" between $\mathcal{P}$ and $\mathcal{NP}$-complete. But are there any *natural* decision[5] problems that fall between these classes? We know only of very precious few candidates: those on the list below (some of which were also discussed in Section 3.5) and a handful of others. We discuss each in turn after listing them.

- **Integer factoring.** Given an integer, find its prime factors (a decision version might ask for the ith bit of the jth prime).

[4] We oversimplified the figure a bit; technically, we only know that $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$, rather than $\mathcal{PH} \subseteq \#\mathcal{P}$.

[5] We discussed search problems in this gap in Section 4.1.

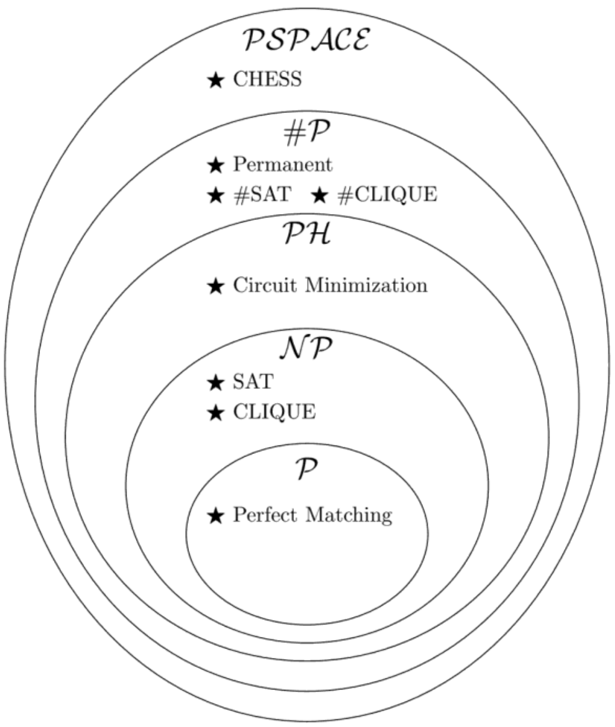


Figure 10. Between $\mathcal{P}$ and $\mathcal{PSPACE}$. As far as we know, all these classes may be equal.

- **Stochastic games.** Three players, White, Black, and Nature, move a token on a directed graph, whose vertices are labeled with players' names. At every step, the token can be moved by the player labeling the vertex it occupies to another along an edge out of that vertex. Nature's moves are random, while White and Black play strategically. Given a labeled graph, and start and target nodes for the token, does White have a strategy that guarantees that the token reaches the target with probability $\geq 1/2$?

- **Knot triviality.** Given a diagram describing a knot (see, e.g., Figure 1), is it the trivial knot?

- **Approximate shortest lattice vector.** Given a (basis for a) lattice L in $\mathbb{R}^n$ and an integer k, does the shortest vector of L have (Euclidean) length at most k, or at least kn? (It is guaranteed that this minimum length is not in $[k, kn]$.)

- **Graph isomorphism.** Given two graphs, are they isomorphic? Namely, is there a bijection between their vertices that preserves the edges?[6]

[6]The recent breakthrough of Babai [Bab15] gives a quasipolynomial-time algorithm for this problem (namely of complexity roughly $\exp((\log n)^{O(1)})$), bringing it very close to $\mathcal{P}$. See also the exposition [HBD17] of this result.

- **Circuit minimization.** The notions appearing in this problem description will be formalized in Chapter 5. Intuitively, it asks for the fastest program computing a function on fixed-size inputs. More formally, given a truth table of a Boolean function f and an integer s, does there exist a Boolean circuit of size at most s computing f? Some evidence of the "intermediate status" of this problem can be found in [AH17] and its references.

Currently we cannot rule out that efficient algorithms will be found for any of these problems, and so some may actually be in $\mathcal{P}$. But we have good formal reasons to believe that they are not $\mathcal{NP}$-complete. This is interesting; we already saw that if a problem is $\mathcal{NP}$-complete, it is an indication that it is not *easy* (namely, in $\mathcal{P}$), if we believe that $\mathcal{P} \neq \mathcal{NP}$. What indications do we have that a problem is *not* universally *hard* (namely, that it is not $\mathcal{NP}$-complete)? Well, if, for example, the problem is in $\mathcal{NP} \cap \text{co}\mathcal{NP}$, and we believe $\mathcal{NP} \neq \text{co}\mathcal{NP}$, then that problem cannot be $\mathcal{NP}$-complete. In both arguments above, unlikely collapses of complexity classes ($\mathcal{P} = \mathcal{NP}$ and $\mathcal{NP} = \text{co}\mathcal{NP}$) give us (perhaps with different confidence levels) an indication as to the complexity of specific problems. This is somewhat satisfactory, in the absence of a definite theorem about their complexity. In particular, we can now explain better why the problems above are not likely to be $\mathcal{NP}$-complete.

The first four problems are all in $\mathcal{NP} \cap \text{co}\mathcal{NP}$. This is clear for (the decision problem of) factoring. For stochastic games, this result is proved in [Con93]. The lattice problem was resolved in [AR05]. The knottedness problem is special: It is a rare example where both inclusions are highly nontrivial. Membership in $\mathcal{NP}$ was proved in [HLP99] (and again, very differently, in [Lac15]). Membership in co$\mathcal{NP}$ was first proved conditionally on the generalized Riemann hypothesis (GRH) in [Kup14],[7] and only recently a different proof that requires no unproven assumption was given in [Lac16].

Graph isomorphism, while in $\mathcal{NP}$, is not known to be in co$\mathcal{NP}$, and so we cannot use the same logic to rule out its possible $\mathcal{NP}$-completeness directly. However, one can apply very similar logic. Graph nonisomorphism has a different type of short, efficient proof, called *interactive proof*, discussed in Chapter 10. Using this, one can prove that if graph isomorphism is $\mathcal{NP}$-complete, it would yield a surprising collapse of the polynomial time hierarchy $\mathcal{PH}$ (defined in Section 4.1). Of course, with the recent quasi-polynomial-time algorithm for graph isomorphism [Bab15] mentioned above, we have much better reasons to believe that it cannot be $\mathcal{NP}$-complete.

The last problem, circuit minimization (which has several variations), is even more mysterious than the previous four. Numerous papers have been written on "unlikely" consequences of its possible easiness (being in $\mathcal{P}$) and its possible hardness (being $\mathcal{NP}$-complete). A recent survey on the topic is [All17].

Finding other natural examples (or better yet, classes of examples) like these will enhance our understanding of the gap $\mathcal{NP} \setminus \mathcal{P}$. Considering the examples above, we expect that mathematics is a more likely source for them than, say, industry. However, for some large classes of natural problems, we know or believe that they *must* exhibit this dichotomy: Every problem in the class is either in $\mathcal{P}$ or is $\mathcal{NP}$-complete. These are classes of constraint-satisfaction problems, which we discuss next.

[7]It may seem mysterious what the GRH has to do with knots, and I encourage you to look at the paper to find out.