OXFORD

THE NATURE *of* COMPUTATION

Cristopher Moore & Stephan Mertens

THE NATURE OF COMPUTATION

The Nature of Computation

Cristopher Moore

Santa Fe Institute

Stephan Mertens

Otto-von-Guericke University, Magdeburg
and
Santa Fe Institute

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS
Great Clarendon Street, Oxford OX2 6DP

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide in

Oxford New York

Auckland Cape Town Dar es Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in

Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

Oxford is a registered trade mark of Oxford University Press
in the UK and in certain other countries

Published in the United States
by Oxford University Press Inc., New York

First published 2011
Reprinted 2012, 2013 (with corrections), 2014 (with corrections),
2016 (twice, once with corrections), 2017 (with corrections), 2018 (with corrections)

British Library Cataloguing in Publication Data
Data available

Library of Congress Cataloging in Publication Data
Data available

Printed in Great Britain
on acid-free paper by
CPI Group (UK) Ltd, Croydon, CR0 4YY

ISBN 978–0–19–923321–2

9 10 8

For Tracy and Doro

Contents

Figure Credits

Preface

> The familiar essayist didn't speak to the millions; he spoke to *one* reader, as if the two of them were sitting side by side in front of a crackling fire with their cravats loosened, their favorite stimulants at hand, and a long evening of conversation stretching before them. His viewpoint was subjective, his frame of reference concrete, his style digressive, his eccentricities conspicuous, and his laughter usually at his own expense. And though he wrote about himself, he also wrote about a *subject*, something with which he was so familiar, and about which he was often so enthusiastic, that his words were suffused with a lover's intimacy.
>
> Anne Fadiman, *At Large and At Small*

> It is not incumbent upon you to finish the work,
> yet neither are you free to desist from it.
>
> Rabbi Tarfon

The sciences that awe and inspire us deal with fundamentals. Biology tries to understand the nature of life, from its cellular machinery to the gorgeous variety of organisms. Physics seeks the laws of nature on every scale from the subatomic to the cosmic. These questions are among the things that make life worth living. Pursuing them is one of the best things humans do.

The theory of computation is no less fundamental. It tries to understand why, and how, some problems are easy while others are hard. This isn't a question of how fast our computers are, any more than astronomy is the study of telescopes. It is a question about the *mathematical structures* of problems, and how these structures help us solve problems or frustrate our attempts to do so. This leads us, in turn, to questions about the nature of mathematical proof, and even of intelligence and creativity.

Computer science can trace its roots back to Euclid. It emerged through the struggle to build a foundation for mathematics in the early 20th century, and flowered with the advent of electronic computers, driven partly by the cryptographic efforts of World War II. Since then, it has grown into a rich field, full of deep ideas and compelling questions. Today it stands beside other sciences as one of the lenses we use to look at the world. Anyone who truly wants to understand how the world works can no more ignore computation than they can ignore relativity or evolution.

Computer science is also one of the most flexible and dynamic sciences. New subfields like quantum computation and phase transitions have produced exciting collaborations between computer scientists, physicists, and mathematicians. When physicists ask what rules govern a quantum system, computer scientists ask what it can compute. When physicists describe the phase transition that turns water to ice, computer scientists ask whether a similar transition turns problems from easy to hard.

This book was born in 2005 when one of us was approached by a publisher to write a book explaining computational complexity to physicists. The tale grew in the telling, until we decided—with some hubris—to explain it to everyone, including computer scientists. A large part of our motivation was to write the book we would have liked to read. We fell in love with the theory of computation because of the beauty and power of its ideas, but many textbooks bury these ideas under a mountain of formalism. We have not hesitated to present material that is technically difficult when it's appropriate. But at every turn we have tried to draw a clear distinction between deep ideas on the one hand and technical details on the other—just as you would when talking to a friend.

Overall, we have endeavored to write our book with the accessibility of Martin Gardner, the playfulness of Douglas Hofstadter, and the lyricism of Vladimir Nabokov. We have almost certainly failed on all three counts. Nevertheless, we hope that the reader will share with us some of the joy and passion we feel for our adopted field. If we have reflected, however dimly, some of the radiance that drew us to this subject, we are content.

We are grateful to many people for their feedback and guidance: Scott Aaronson, Heiko Bauke, Paul Chapman, Andrew Childs, Aaron Clauset, Varsha Dani, Josep Díaz, Owen Densmore, Irit Dinur, Ehud Friedgut, Tom Hayes, Robert Hearn, Stefan Helmreich, Reuben Hersh, Shiva Kasiviswanathan, Brian Karrer, David Kempe, Greg Kuperberg, Cormac McCarthy, Sarah Miracle, John F. Moore, Michel Morvan, Larry Nazareth, Sebastian Oberhoff, Ryan O'Donnell, Mark Olah, Jim Propp, Dana Randall, Sasha Razborov, Omer Reingold, Paul Rendell, Sara Robinson, Jean-Baptiste Rouquier, Amin Saberi, Jared Saia, Nicolas Schabanel, Cosma Shalizi, Thérèse Smith, Darko Stefanović, John Tromp, Vijay Vazirani, Robin Whitty, Lance Williams, Damien Woods, Jon Yard, Danny Yee, Lenka Zdeborová, Yaojia Zhu, and Katharina Zweig.

We are also grateful to Lee Altenberg, László Babai, Nick Baxter, Nirdosh Bhatnagar, Marcus Calhoun-Lopez, Timothy Chow, Nathan Collins, Alex Conley, Will Courtney, Zheng Cui, Wim van Dam, Tom Dangniam, Aaron Denney, Hang Dinh, David Doty, Taylor Dupuy, Bryan Eastin, Charles Efferson, Veit Elser, Leigh Fanning, Steve Flammia, Matthew Fricke, Michel Goemans, Benjamin Gordon, Stephen Guerin, Samuel Gutierrez, Russell Hanson, Jacob Hobbs, Neal Holtschulte, Peter Høyer, Luan Jun, Valentine Kabanets, Richard Kenyon, Jeffrey Knockel, Leonid Kontorovich, Maurizio Leo, MatjažLeonardis, Phil Lewis, Scott Levy, Chien-Chi Lo, Jun Luan, Shuang Luan, Sebastian Luther, Jon Machta, Jonathan Mandeville, Bodo Manthey, Pierre McKenzie, Pete Morcos, Brian Nelson, ThanhVu Nguyen, Katherine Nystrom, Olumuyiwa Oluwasanmi, Boleszek Osinski, John Patchett, Robin Pemantle, Yuval Peres, Carlos Riofrio, Tyler Rush, Navin Rustagi, George Saad, Gary Sandine, Samantha Schwartz, Oleg Semenov, David Sherrington, Jon Sorenson, George Stelle, Satomi Sugaya, Bert Tanner, Amitabh Trehan, Yamel Torres, Michael Velbaum, Lutz Warnke, Chris Willmore, David Wilson, Chris Wood, Ben Yackley, Yiming Yang, Rich Younger, Sheng-Yang Wang, Zhan Zhang, and Evgeni Zlatanov. We apologize for any omissions.

We express our heartfelt thanks to the Santa Fe Institute, without whose hospitality we would have been unable to complete this work. In particular, the SFI library staff—Margaret Alexander, Tim Taylor, and Joy LeCuyer—fulfilled literally hundreds of requests for articles and books.

We are grateful to our editor Sönke Adlung for his patience, and to Alison Lees for her careful copyediting. Mark Newman gave us invaluable help with the LaTeX Memoir class, in which this book is typeset, along with insights and moral support from his own book-writing experiences. And throughout the process, Alex Russell shaped our sense of the field, separating the wheat from the chaff and helping us to decide which topics and results to present to the reader. Some fabulous monsters didn't make it onto the ark, but many of those that did are here because he urged us to take them on board.

Finally, we dedicate this book to Tracy Conrad and Doro Frederking. Our partners, our loves, they have made everything possible.

Cristopher Moore and Stephan Mertens
Santa Fe and Magdeburg, 2019

How to read this book

> Outside a dog a book is a man's best friend.
> Inside a dog it's too dark to read.
>
> Groucho Marx

We recommend reading Chapters 1–7 in linear order, and then picking and choosing from later chapters and sections as you like. Even the advanced chapters have sections that are accessible to nearly everyone.

For the most part, the only mathematics we assume is linear algebra and some occasional calculus. We use Fourier analysis and complex numbers in several places, especially Chapter 11 for the PCP Theorem and Chapter 15 on quantum computing. Mathematical techniques that we use throughout the book, such as asymptotic notation and discrete probability, are discussed in the Appendix. We assume some minimal familiarity with programming, such as the meaning of **for** and **while** loops.

Scattered throughout the text you will find Exercises. These are meant to be easy, and to help you check whether you are following the discussion at that point. The Problems at the end of each chapter delve more deeply into the subject, providing examples and fleshing out arguments that we sketch in the main text. We have been generous with hints and guideposts in order to make even the more demanding problems doable.

Every once in a while, you will see a quill symbol in the margin—yes, like that one there. This refers to a note in the Notes section at the end of the chapter, where you can find details, historical discussion, and references to the literature.

Since theoretical computer science is a rapidly evolving field, we invite the reader to check periodically for updates and addenda at www.nature-of-computation.org. There we will include new problems and exercises, and notes and references for new results.

A note to the instructor

We have found that Chapters 1–8, with selections from Chapters 9–11, form a good text for an introductory graduate course on computational complexity. We and others have successfully used later chapters as texts or supplementary material for more specialized courses, such as Chapters 12 and 13 for a course on Markov chains, Chapter 14 for phase transitions, and Chapter 15 for quantum computing. Some old-fashioned topics, like formal languages and automata, are missing from our book, and this is by design.

The Turing machine has a special place in the history of computation, and we discuss it along with λ-calculus and partial recursive functions in Chapter 7. But we decided early on to write about computation as if the Church-Turing thesis were true—in other words, that we are free to use whatever model of computation makes it easiest to convey the key ideas. Accordingly, we describe algorithms at a "software" level, as programs written in the reader's favorite programming language. This lets us draw on the reader's experience and intuition that programs need time and memory to run. Where necessary, such as in our discussion of LOGSPACE in Chapter 8, we drill down into the hardware and discuss details such as our model of memory access.

Please share with us your experiences with the book, as well as any mistakes or deficiencies you find. We maintain errata at www.nature-of-computation.org. We can also provide a solution manual on request, which currently contains solutions for over half of the problems.

Note on the 2018 printing

This printing corrects a large number of typos throughout the book, fixes a number of aesthetic typesetting issues, and updates many of the Notes to take recent advances into account. With the help of our dedicated readers, we have also improved the discussion and readability of the book and corrected some mathematical errors, for instance in the proof that quadratic Diophantine equations are NP-complete (Section 5.4.4 and Problems 5.24 and 5.25). There are a number of new problems, added at the end of the Problems sections so as not to disturb the numbering of previous ones. In particular, readers interested in pseudorandomness and derandomization will be pleased to learn that Nisan's generator for fooling space-bounded computation now appears, with many hints to guide the reader, as Problem 11.18.

Chapter 1

Prologue

Some citizens of Königsberg
Were walking on the strand
Beside the river Pregel
With its seven bridges spanned.

"O Euler, come and walk with us,"
Those burghers did beseech.
"We'll roam the seven bridges o'er,
And pass but once by each."

"It can't be done," thus Euler cried.
"Here comes the Q.E.D.
Your islands are but vertices
And four have odd degree."

William T. Tutte

1.1 Crossing Bridges

We begin our journey into the nature of computation with a walk through 18th-century Königsberg (now Kaliningrad). As you can see from Figure 1.1, the town of Königsberg straddles the river Pregel with seven bridges, which connect the two banks of the river with two islands. A popular puzzle of the time asked if one could walk through the city in a way that crosses each bridge exactly once. We do not know how hard the burghers of Königsberg tried to solve this puzzle on their Sunday afternoon walks, but we do know that they never succeeded.

It was Leonhard Euler who solved this puzzle in 1736. As a mathematician, Euler preferred to address the problem by pure thought rather than by experiment. He recognized that the problem depends only on the set of connections between the riverbanks and islands—a *graph* in modern terms. The graph corresponding to Königsberg has four vertices representing the two riverbanks and the two islands, and seven edges for the bridges, as shown in Figure 1.2. Today, we say that a walk through a graph that crosses


1.1

FIGURE 1.1: Königsberg in the 17th century.

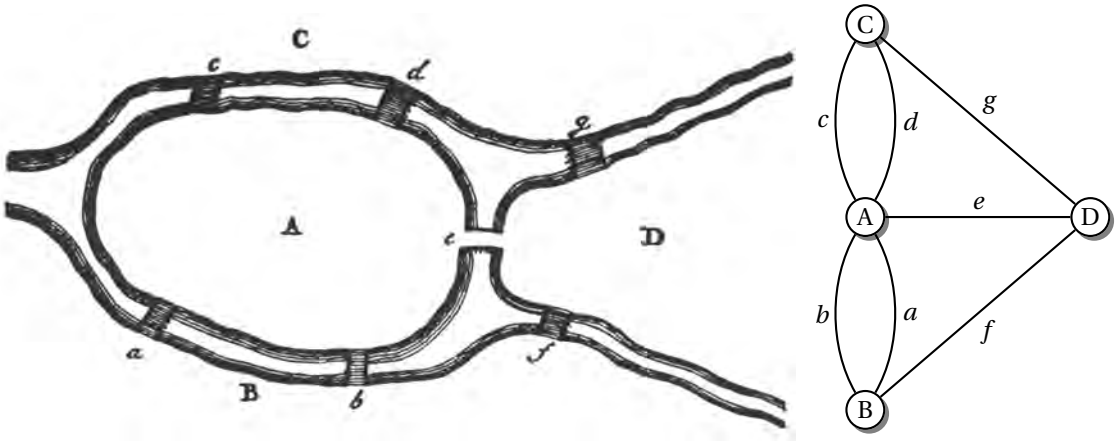


FIGURE 1.2: The seven bridges of Königsberg. Left, as drawn in Euler's 1736 paper, and right, as represented as a graph in which each riverbank or island is a vertex and each bridge an edge.

each edge once is an *Eulerian path*, or an *Eulerian cycle* if it returns to its starting point. We say that a graph is Eulerian if it possesses an Eulerian cycle.

Now that we have reduced the problem to a graph that we can doodle on a sheet of paper, it is easy to explore various walks by trial and error. Euler realized, though, that trying all possible walks this way would take some time. As he noted in his paper (translated from the Latin):

> As far as the problem of the seven bridges of Königsberg is concerned, it can be solved by making an exhaustive list of possible routes, and then finding whether or not any route satisfies the conditions of the problem. Because of the number of possibilities, this method of solutions would be too difficult and laborious, and in other problems with more bridges, it would be impossible.

Let's be more quantitative about this. Assume for simplicity that each time we arrive on an island or a riverbank there are two different ways we could leave. Then if there are n bridges to cross, a rough estimate for the number of possible walks would be 2^n. In the Königsberg puzzle we have $n = 7$, and while 2^7 or 128 routes would take quite a while to generate and check by hand, a modern computer could do so in the blink of an eye.

But Euler's remark is not just about the bridges of Königsberg. It is about the entire *family of problems of this kind*, and how their difficulty grows, or *scales*, as a function of the number of bridges. If we consider the bridges of Venice instead, where $n = 420$—or Pittsburgh, where $n = 446$—even the fastest computer imaginable would take longer than the age of the universe to do an exhaustive search. Thus, if searching through the space of all possible solutions were the only way to solve these problems, even moderately large cities would be beyond our computational powers.

Euler had a clever insight which allows us to avoid this search completely. He noticed that in order to cross each edge once, any time we arrive at a vertex along one edge, we have to depart on a different edge. Thus the edges of each vertex must come in pairs, with a "departure" edge for each "arrival" edge. It follows that the *degree* of each vertex—that is, the number of edges that touch it—must be even. This

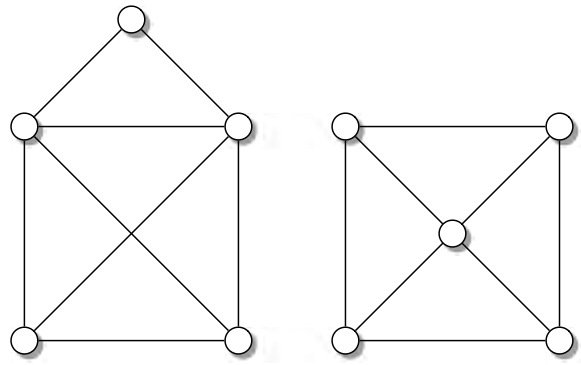


FIGURE 1.3: A classic children's puzzle. Can you draw these graphs without lifting the pen from the paper, or drawing the same edge twice? Equivalently, do they have Eulerian paths?

holds for all vertices except the vertices where the path starts and ends, which must have odd degree unless they coincide and the path is a cycle.

This argument shows that a necessary condition for a graph to be Eulerian is for all its vertices to have even degree. Euler claimed that this condition is also sufficient, and stated the following theorem:

Theorem 1.1 *A connected graph contains an Eulerian cycle if and only if every vertex has even degree. If exactly two vertices have odd degree, it contains an Eulerian path but not an Eulerian cycle.*

This theorem allows us to solve the bridges of Königsberg very quickly. As the poem at the head of this chapter points out, all four vertices have odd degree, so there is no Eulerian path through the old town of Königsberg.

Beyond solving this one puzzle, Euler's insight makes an enormous difference in how the complexity of this problem scales. An exhaustive search in a city with n bridges takes an amount of time that grows *exponentially* with n. But we can check that every vertex has even degree in an amount of time proportional to the number of vertices, assuming that we are given the map of the city in some convenient format. Thus Euler's method lets us solve this problem in *linear* time, rather than the exponential time of a brute-force search. Now the bridges of Venice, and even larger cities, are easily within our reach.

Exercise 1.1 *Which of the graphs in Figure 1.3 have Eulerian paths?*

In addition to the tremendous speedup from exponential to linear time, Euler's insight transforms this problem in another way. What does it take to *prove* the existence, or nonexistence, of an Eulerian path? If one exists, we can easily prove this fact simply by exhibiting it. But if no path exists, how can we prove that? How can we convince the people of Königsberg that their efforts are futile?

Imagine what would have happened if Euler had used the brute-force approach, presenting the burghers with a long list of all possible paths and pointing out that none of them work. Angry at Euler for spoiling their favorite Sunday afternoon pastime, they would have been understandably skeptical. Many would have refused to go through the tedious process of checking the entire list, and would have held out

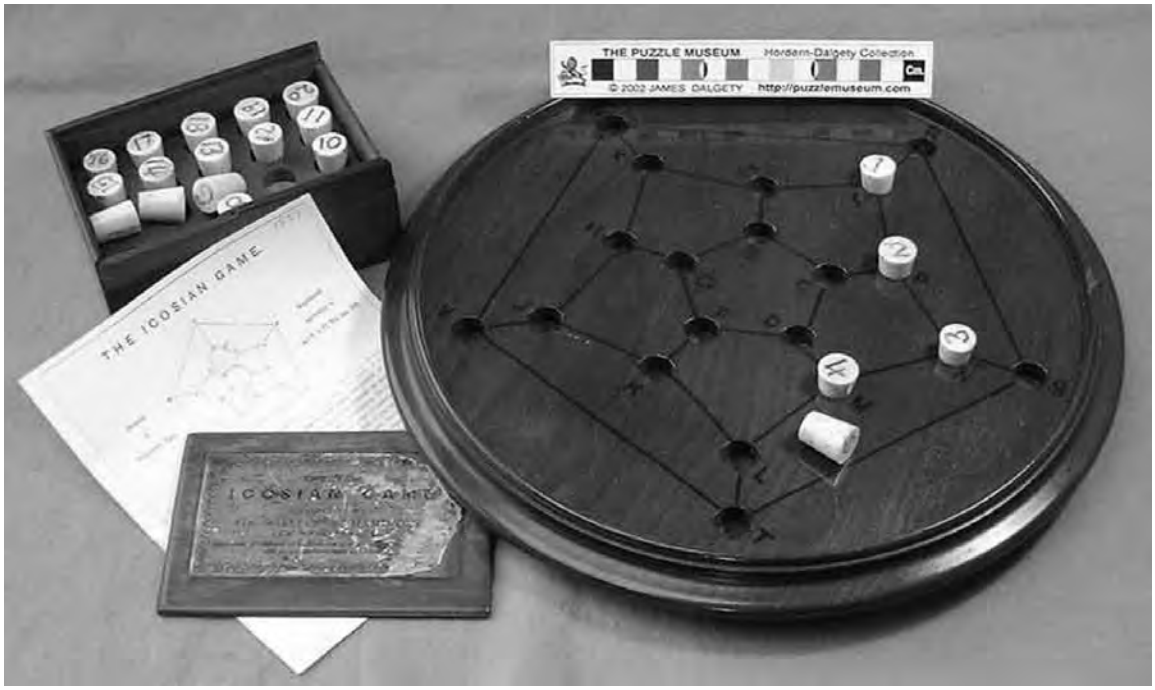


FIGURE 1.4: Hamilton's Icosian game.

the hope that Euler had missed some possibility. Moreover, such an ungainly proof is rather unsatisfying, even if it is logically airtight. It offers no sense of *why* no path exists.

1.2

In contrast, even the most determined skeptic can follow the argument of Euler's theorem. This allows Euler to present a proof that is simple, compact, and irresistible: he simply needs to exhibit three vertices with odd degree. Thus, by showing that the existence of a path is equivalent to a much simpler property, Euler radically changed the *logical structure* of the problem, and the type of proof or disproof it requires.

1.2 Intractable Itineraries

The next step in our journey brings us to 19th-century Ireland and the Astronomer Royal, Sir William Rowan Hamilton, known to every physicist through his contributions to classical mechanics. In 1859, Hamilton put a new puzzle on the market, called the "Icosian game," shown in Figure 1.4. The game was a commercial failure, but it led to one of the iconic problems in computer science today.

The object of the game is to walk around the edges of a dodecahedron while visiting each vertex once and only once. Actually, it was a two-player game in which one player chooses the first five vertices, and the other tries to complete the path—but for now, let's just think about the solitaire version. While such walks had been considered in other contexts before, we call them *Hamiltonian* paths or cycles today, and

1.3

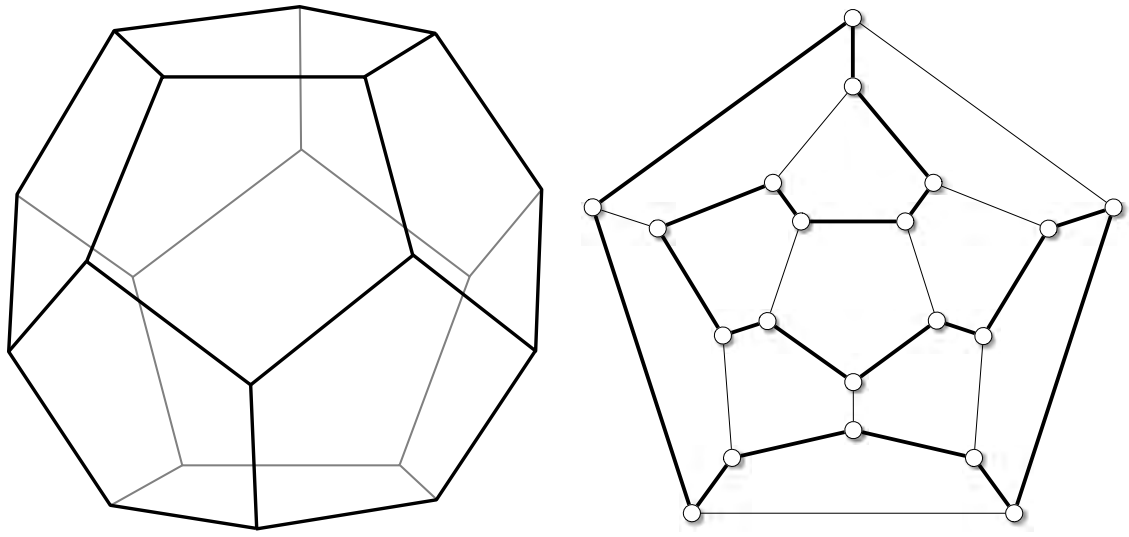


FIGURE 1.5: Left, the dodecahedron; right, a flattened version of the graph formed by its edges. One Hamiltonian cycle, which visits each vertex once and returns to its starting point, is shown in bold.

we say that a graph is Hamiltonian if it possesses a Hamiltonian cycle. One such cycle for the dodecahedron is shown in Figure 1.5.

At first Hamilton's puzzle seems very similar to the bridges of Königsberg. Eulerian paths cross each edge once, and Hamiltonian paths visit each vertex once. Surely these problems are not very different? However, while Euler's theorem allows us to avoid a laborious search for Eulerian paths or cycles, we have no such insight into Hamiltonian ones. As far as we know, there is no simple property—analogous to having vertices of even degree—to which *Hamiltonianness* is equivalent.

As a consequence, we know of no way of avoiding, essentially, an exhaustive search for Hamiltonian paths. We can visualize this search as a tree as shown in Figure 1.6. Each node of the tree corresponds to a partial path, and branches into child nodes corresponding to the various ways we can extend the path. In general, the number of nodes in this search tree grows exponentially with the number of vertices of the underlying graph, so traversing the entire tree—either finding a leaf with a complete path, or learning that every possible path gets stuck—takes exponential time.

To phrase this computationally, we believe that there is no program, or *algorithm*, that tells whether a graph with n vertices is Hamiltonian or not in an amount of time proportional to n, or n^2, or any polynomial function of n. We believe, instead, that the best possible algorithm takes exponential time, 2^{cn} for some constant $c > 0$. Note that this is not a belief about how fast we can make our computers. Rather, it is a belief that finding Hamiltonian paths is *fundamentally harder* than finding Eulerian ones. It says that these two problems differ in a deep and qualitative way.

While finding a Hamiltonian path seems to be hard, *checking* whether a given path is Hamiltonian is easy. Simply follow the path vertex by vertex, and check that it visits each vertex once. So if a computationally powerful friend claims that a graph has a Hamiltonian path, you can challenge him or her to

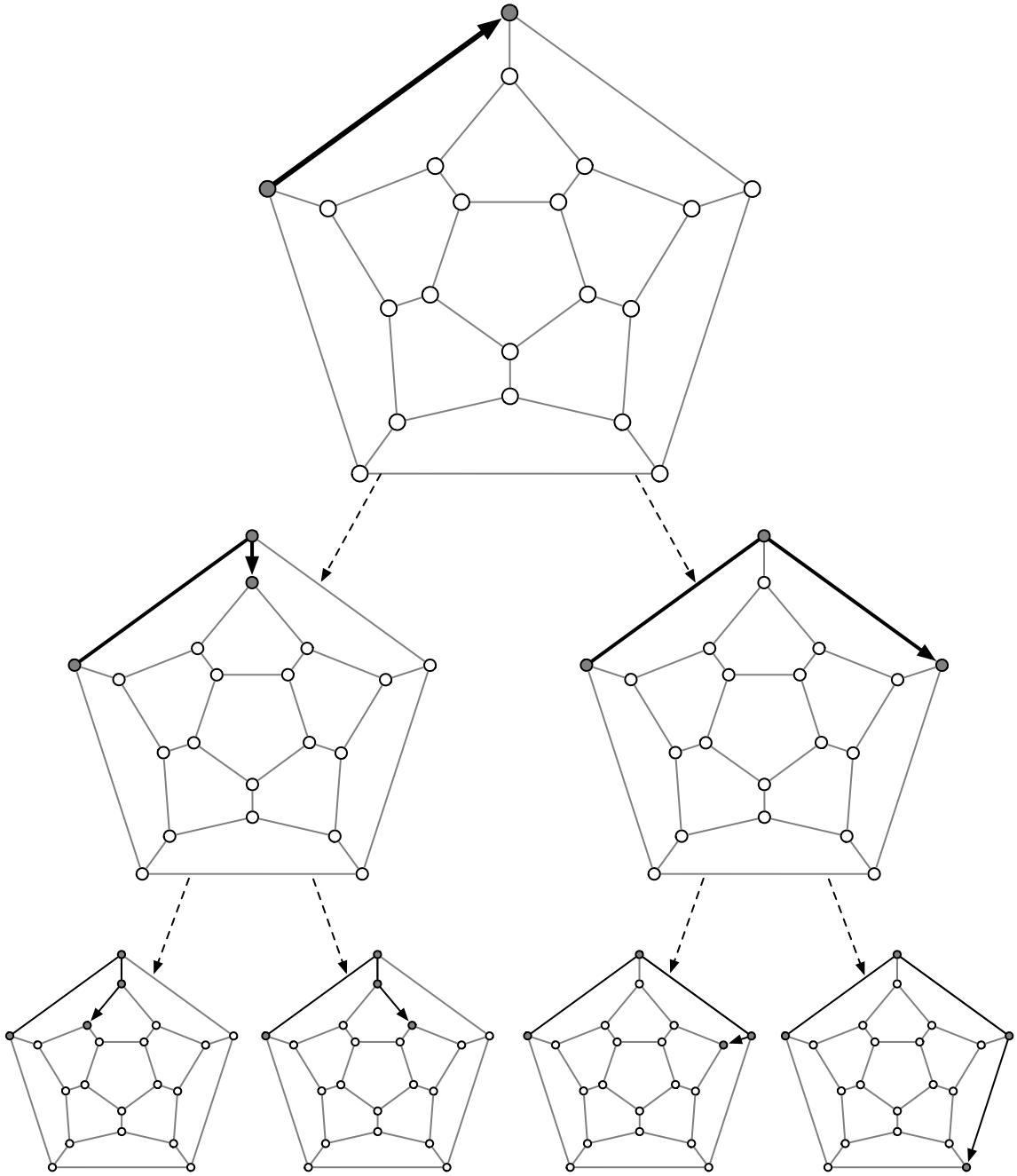


FIGURE 1.6: The first two levels of the search tree for a Hamiltonian path.

prove that fact by showing it to you, and you can then quickly confirm or refute their claim. Problems like these are rather like finding a needle in a haystack: if I show you a needle you can confirm that it is one, but it's hard to find a needle—at least without a magnetic insight like Euler's.

On the other hand, if I claim that a haystack has no needles in it, the only way to prove this is to sift carefully through all the hay. Similarly, we know of no way to prove that no Hamiltonian cycle exists without a massive search. Unlike Eulerian cycles, where there is a simple proof in either case, the logical structure of Hamiltonianness seems to be fundamentally asymmetric—proving the nonexistence of a Hamiltonian cycle seems to be much harder than proving its existence.

Of course one might think that there is a more efficient method for determining whether a graph is Hamiltonian, and that we have simply not been clever enough to find it. But as we will see in this book, there are very good reasons to believe that *no such method exists*. Even more amazingly, if we are wrong about this—if Hamilton's problem can be solved in time that only grows polynomially—then so can thousands of other problems, all of which are currently believed to be exponentially hard. These problems range from such classic search and optimization problems as the Traveling Salesman problem, to the problem of finding short proofs of the grand unsolved questions in mathematics. In a very real sense, the hardness of Hamilton's problem is related to our deepest beliefs about mathematical and scientific creativity. Actually *proving* that it is hard remains one of the holy grails of theoretical computer science.

1.3 Playing Chess With God

> It's a sort of Chess that has nothing to do with Chess, a Chess that we could never have imagined without computers. The Stiller moves are awesome, almost scary, because you know they are the truth, God's Algorithm—it's like being revealed the Meaning of Life, but you don't understand one word.
>
> Tim Krabbé

As we saw in the previous section, the problem of telling whether a Hamiltonian cycle exists has the property that finding solutions is hard—or so we believe—but checking them is easy. Another example of this phenomenon is factoring integers. As far as we know, there is no efficient way to factor a large integer N into its divisors—at least without a quantum computer—and we base the modern cryptosystems used by intelligence agents and Internet merchants on this belief. On the other hand, given two numbers p and q it is easy to multiply them, and check whether $pq = N$.

1.4

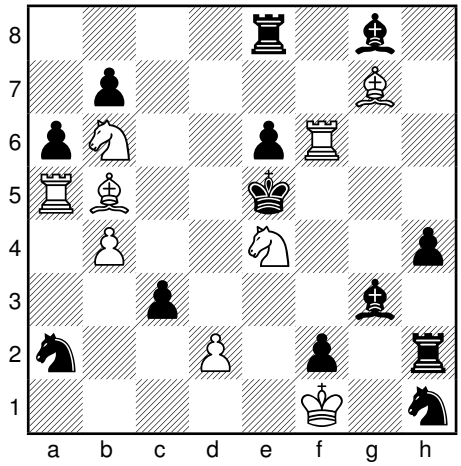
This fact was illustrated beautifully at a meeting of the American Mathematical Society in 1903. The mathematician Frank Nelson Cole gave a "lecture without words," silently performing the multiplication

$$193\,707\,721 \times 761\,838\,257\,287 = 147\,573\,952\,588\,676\,412\,927$$

on the blackboard. The number on the right-hand side is $2^{67} - 1$, which the 17th-century French mathematician Marin Mersenne conjectured is prime. In 1876, Édouard Lucas managed to prove that it is composite, but gave no indication of what numbers would divide it. The audience, knowing full well how hard it is to factor 21-digit numbers, greeted Cole's presentation with a standing ovation. Cole later admitted that it had taken him "three years of Sundays" to find the factors.

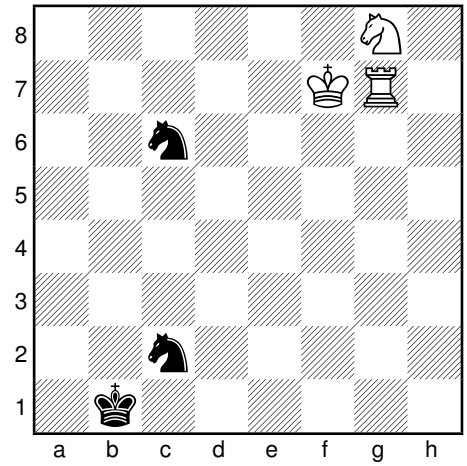
On the other hand, there are problems for which even *checking* a solution is extremely hard. Consider the Chess problems shown in Figure 1.7. Each claims that White has a winning strategy, which will lead

Sam Loyd (1903)



Mate in 3

Lewis Stiller (1995)



Mate in 262

FIGURE 1.7: Chess problems are hard to solve—and hard to check.

inexorably to checkmate after n moves. On the left, $n = 3$, and seeing how to corner Black—after a very surprising first move—is the work of a pleasant afternoon. On the right, we have a somewhat larger value of n: we claim that White can force Black into checkmate after 262 moves.

But how, dear reader, can we prove this claim to you? Unlike the problems of the previous two sections, we are no longer playing solitaire: we have an opponent who will do their best to win. This means that it's not enough to prove the existence of a simple object like a Hamiltonian path. We have to show that there exists a move for White, such that no matter how Black replies, there exists a move for White, such that no matter how Black replies, and so on... until, at most 262 moves later, every possible game ends in checkmate for Black. As we go forward in time, our opponent's moves cause the game to branch into a exponential tree of possibilities, and we have to show that a checkmate awaits at every leaf. Thus a strategy is a much larger object, with a much deeper logical structure, than a path.

There is indeed a proof, consisting of a massive database of endgames generated by a computer search, that White can mate Black in 262 moves in the position of Figure 1.7. But verifying this proof far exceeds the capabilities of human beings, since it requires us to check every possible line of play. The best we can do is look at the program that performed the search, and convince ourselves that it will run correctly. As of 2007, an even larger search has confirmed the long-standing opinion of human players that Checkers is a draw under perfect play. For humans with our finite abilities, however, Chess and Checkers will always keep their secrets.

1.5

1.4 What Lies Ahead

As we have seen with our three examples, different problems require fundamentally different kinds of search, and different types of proof, to find or verify their solutions. Understanding how to solve problems as efficiently as possible—and understanding how, and why, some problems are extremely hard—is the subject of our book. In the chapters ahead, we will ask, and sometimes answer, questions like the following:

- Some problems have insights like Euler's, and others seem to require an exhaustive search. What makes the difference? What kinds of strategies can we use to skip or simplify this search, and for which problems do they work?

- A host of problems—finding Hamiltonian paths, coloring graphs, satisfying formulas, and balancing numbers—are all equally hard. If we could solve any of them efficiently, we could solve all of them. What do these problems have in common? How can we transform one of them into the other?

- If we could find Hamiltonian paths efficiently, we could also easily find short proofs—if they exist—of the great unsolved problems in mathematics, such as the Riemann Hypothesis. We believe that doing mathematics is harder than this, and that it requires all our creativity and intuition. But does it really? Can we prove that finding proofs is hard?

- Can one programming language, or kind of computer, solve problems that another can't? Or are all sufficiently powerful computers equivalent? Are even simple systems, made of counters, tiles, and billiard balls, capable of universal computation?

- Are there problems that no computer can solve, no matter how much time we give them? Are there mathematical truths that no axiomatic system can prove?

- If exact solutions are hard to find, can we find approximate ones? Are there problems where even approximate solutions are hard to find? Are there others that are hard to solve perfectly, but where we can find solutions that are as close as we like to the best possible one?

- What happens if we focus on the amount of memory a computation needs, rather than the time it takes? How much memory do we need to find our way through a maze, or find a winning strategy in a two-player game?

- If we commit ourselves to one problem-solving strategy, a clever adversary can come up with the hardest possible example. Can we defeat the adversary by acting unpredictably, and flipping coins to decide what to do?

- Suppose that Merlin has computational power beyond our wildest dreams, but that Arthur is a mere mortal. If Merlin knows that White has a winning strategy in Chess, can he convince Arthur of that fact, without playing a single game? How much can Arthur learn by asking Merlin random questions? What happens if Merlin tries to deceive Arthur, or if Arthur tries to "cheat" and learn more than he was supposed to?

- If flipping random coins helps us solve problems, do we need truly random coins? Are there strong pseudorandom generators—fast algorithms that produce strings of coin flips deterministically, but with no pattern that other fast algorithms can discover?
- Long proofs can have small mistakes hidden in them. But are there "holographic" proofs, which we can confirm are almost certainly correct by checking them in just a few places?
- Finding a solution to a problem is one thing. What happens if we want to generate a random solution, or count the number of solutions? If we take a random walk in the space of all possible solutions, how long will it take to reach an equilibrium where all solutions are equally likely?
- How rare are the truly hard examples of hard problems? If we make up random examples of a hard problem, are they hard or easy? When we add more and more constraints to a problem in a random way, do they make a sudden jump from solvable to unsolvable?
- Finally, how will quantum computers change the landscape of complexity? What problems can they solve faster than classical computers?

Problems

> A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem may be modest, but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery.
>
> George Pólya, *How To Solve It*

1.1 Handshakes. Prove that in any finite graph, the number of vertices with odd degree is even.

1.2 Pigeons and holes. Properly speaking, we should call our representation of Königsberg a *multigraph*, since some pairs of vertices are connected to each other by more than one edge. A *simple* graph is one in which there are no multiple edges, and no self-loops.

Show that in any finite simple graph with more than one vertex, there is at least one pair of vertices that have the same degree. Hint: if n pigeons try to nest in $n-1$ holes, at least one hole will contain more than one pigeon. This simple but important observation is called the *pigeonhole principle*.

1.3 Proving Euler's claim. Euler didn't actually prove that having vertices with even degree is sufficient for a connected graph to be Eulerian—he simply stated that it is obvious. This lack of rigor was common among 18th century mathematicians. The first real proof was given by Carl Hierholzer more than 100 years later. To reconstruct it, first show that if every vertex has even degree, we can cover the graph with a set of cycles such that every edge appears exactly once. Then consider combining cycles with moves like those in Figure 1.8.

1.4 Finding an Eulerian path. Let's turn the proof of the previous problem into a simple algorithm that constructs an Eulerian path. If removing an edge will cause a connected graph to fall apart into two pieces, we call that edge a *bridge*. Now consider the following simple rule, known as Fleury's algorithm: at each step, consider the graph G' formed by the edges you have not yet crossed, and only cross a bridge of G' if you have to. Show that if a connected graph has two vertices of odd degree and we start at one of them, this algorithm will produce an Eulerian path, and that if all vertices have even degree, it will produce an Eulerian cycle no matter where we start.

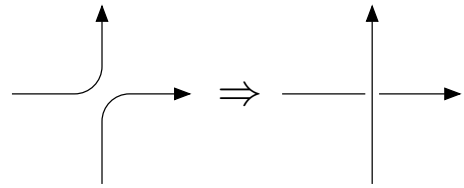


FIGURE 1.8: Combining cycles at a crossing.

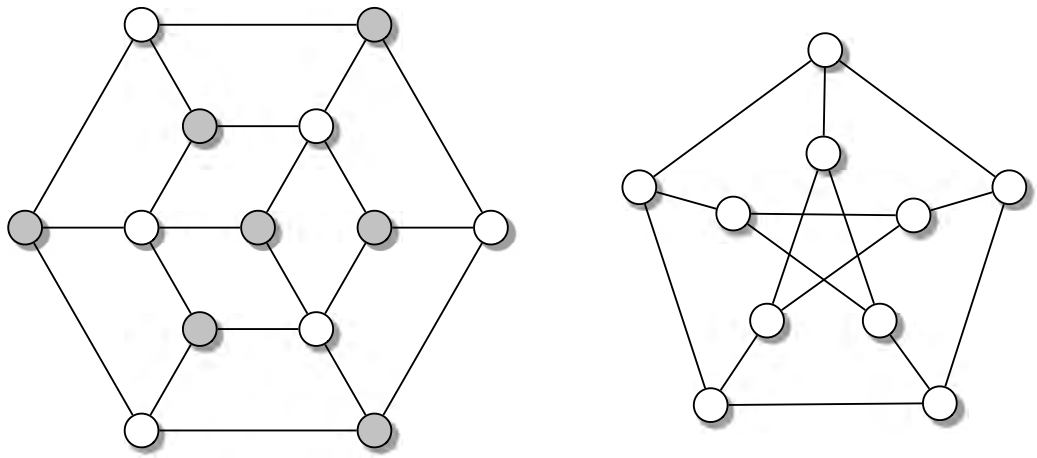


FIGURE 1.9: Two graphs that have Hamiltonian paths, but not Hamiltonian cycles.

1.5 One-way bridges. A *directed graph* is one where each edge has an arrow on it. Thus there can be an edge from u to v without one from v to u, and a given vertex can have both incoming and outgoing edges. An Eulerian path would be one that crosses each edge once, moving in the direction allowed by the arrow. Generalize Euler's theorem by stating under what circumstances a directed graph is Eulerian.

1.6 Plato and Hamilton. Inspired by Hamilton's choice of the dodecahedron, consider the other four Platonic solids, and the graphs consisting of their corners and edges: the tetrahedron, cube, octahedron, and icosahedron. Which ones are Hamiltonian? Which are Eulerian?

1.7 A rook's tour. Let G be an $m \times n$ grid—that is, a graph with mn vertices arranged in an $m \times n$ rectangle, with each vertex connected to its nearest neighbors. Assume that $m, n > 1$. Prove that G is Hamiltonian if either m or n is even, but not if both m and n are odd.

1.8 Symmetry and parity. Show that each graph in Figure 1.9 has a Hamiltonian path, but no Hamiltonian cycle. Hint: use the colors for the one on the left. For the one on the right, called the Petersen graph, exploit its symmetry.

1.9 The Chinese postman. The *Chinese postman problem*, named in honor of the Chinese mathematician Mei-Ko Kwan, asks for the shortest cyclic tour of a graph that crosses every edge *at least* once. If the graph is not Eulerian, the postman has to repeat some edges. Show that the shortest postman's tour crosses each edge at most twice, and

that this worst case only occurs if the graph is a tree. Show, moreover, that the repeated edges form a set of paths that connect pairs of vertices of odd degree, creating a *matching* where each odd-degree vertex has another one as a partner. It turns out (see Note 5.6) that there is a polynomial-time algorithm that finds the matching that minimizes the total length of these paths, and hence the shortest postman's tour.

1.10 Existence, search, and the oracle. We can consider two rather different problems regarding Hamiltonian cycles. One is the *decision problem*, the yes-or-no question of whether such a cycle exists. The other is the *search problem* or *function problem*, in which we want to actually find the cycle.

Suppose that there is an oracle in a nearby cave. She will tell us, for the price of one drachma per question, whether a graph has a Hamiltonian cycle. If it does, show that by asking her a series of questions, perhaps involving modified versions of the original graph, we can find the Hamiltonian cycle after spending a number of drachmas that grows polynomially as a function of the number of vertices. Thus if we can solve the decision problem in polynomial time, we can solve the search problem as well.

Notes

1.1 Graph theory. Euler's paper on the Königsberg bridges [272] can be regarded as the birth of graph theory, the mathematics of connectivity. The translation we use here appears in [111], which contains many of the key early papers in this field. Today graph theory is a very lively branch of discrete mathematics with many applications in chemistry, engineering, physics, and computer science.

We will introduce concepts from graph theory "on the fly," as we need them. For an intuitive introduction, we recommend Trudeau's little treatise [796]. For a more standard textbooks, see Bollobás [119] or the *Handbook on Graph Theory* [354]. Hierholzer's proof, which we ask you to reconstruct in Problem 1.3, appeared in [400]. The Chinese Postman of Problem 1.9 was studied by Mei-Ko Kwan in 1962 [512].

The river Pregel still crosses the city of Kaliningrad, but the number of bridges and their topology changed considerably during World War II. See also [614].

1.2 Persuasive proofs. According to the great Hungarian mathematician Paul Erdős, God has a book that contains short and insightful proofs of every theorem, and sometimes humans can catch a glimpse of a few of its pages. One of the highest forms of praise that one mathematician can give another is to say "Ah, you have found the proof from the book." [26].

In a sense, this is the difference between Eulerian and Hamiltonian paths. If a large graph G lacks an Eulerian path, Euler's argument gives a "book proof" of that fact. In contrast, as far as we know, the only way to prove that G lacks a Hamiltonian path is an ugly exhaustive search.

This dichotomy exists in other areas of mathematics as well. A famous example is Thomas Hales' proof of the Kepler conjecture [362], a 400-year-old claim about the densest packings of spheres in three-dimensional space. After four years of work, a group of twelve referees concluded that they were "99% certain" of the correctness of the proof. Hales' proof relies on exhaustive case checking by a computer, a technique first used in the proof of the Four Color Theorem. Such a proof may confirm that a fact is true, but it offers very little illumination about *why* it is true. See [220] for a thought-provoking discussion of the complexity of mathematical proofs, and how computer-assisted proofs will change the nature of mathematics.

1.3 A knight's tour. One instance of the Hamiltonian path problem is far older than the Icosian game. Around the year 840 A.D., a Chess player named al-Adli ar-Rumi constructed a *knight's tour* of the chessboard, shown in Figure 1.10. We can think of this as a Hamiltonian path on the graph whose vertices are squares, and where two vertices are adjacent if a knight could move from one to the other. Another early devotee of knight's tours was the Kashmiri poet Rudrata (circa 900 A.D.). In their lovely book *Algorithms* [215], Dasgupta, Papadimitriou and Vazirani suggest that Hamiltonian paths be called Rudrata paths in his honor.

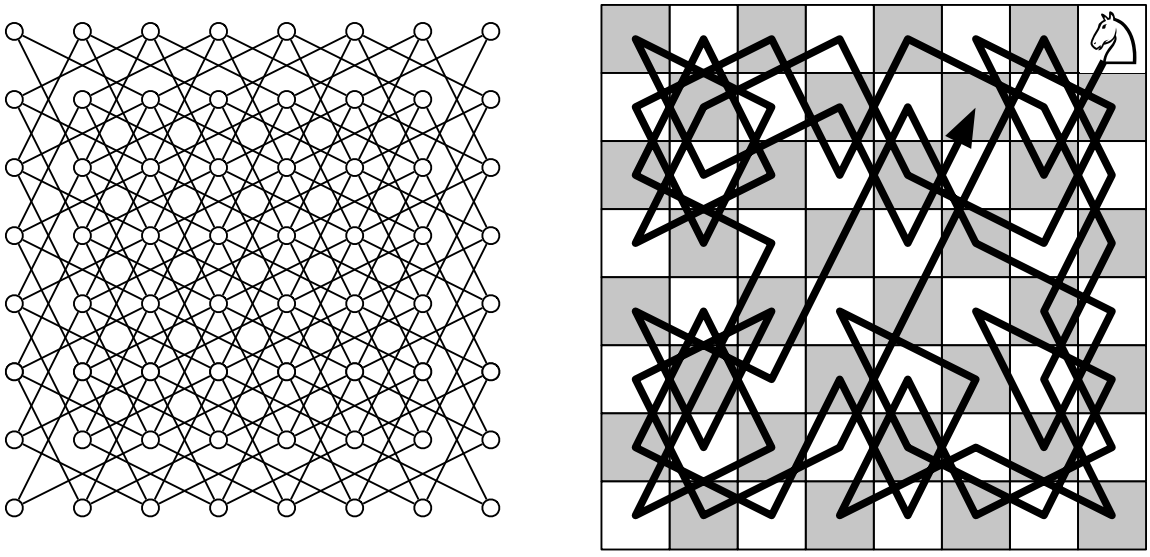


FIGURE 1.10: On the left, the graph corresponding to the squares of the chessboard with knight's moves between them. On the right, Al-Adli's Hamiltonian tour, or knight's tour, of this graph. Note that with one more move it becomes a Hamiltonian cycle.

1.4 Cryptic factors. As we will see in Chapter 15, it is not quite right to say that cryptosystems like RSA public-key encryption are based on the hardness of factoring: they are based on other number-theoretic problems, which might be easier than factoring. However, if we can factor large integers efficiently, we can solve these problems too.

1.5 Endgame. The *endgame tablebase* is a computerized database of all endgame positions in Chess with a given number of pieces, that reveals the value (win, loss or draw) of each position and and how many moves it will take to achieve that result with perfect play [789]. The 262-move problem in Figure 1.7 was found by Stiller [774], who extended the tablebase to six-piece endgames. Of all positions with the given material that are a win, it has the longest "distance to mate."

The proof that Checkers is a draw [729] uses a similar endgame database that contains the value of all Checkers positions with ten pieces or fewer. There are 3.9×10^{13} of these positions, compressed into 237 gigabytes of diskspace. The proof traces all relevant lines of play from the starting position to one of the positions in this database. This proof constitutes what game theorists call a *weak* solution of a game. For a *strong* solution, we would have to compute the optimal move for every legal position on the board, not just the opening position. We will discuss the enormous difficulty of this problem in Chapter 8.

Can we hope for a weak solution of Chess? The six-piece endgame database for Chess is more than five times bigger than the ten-piece database for Checkers. This is no surprise since Checkers uses only half of the squares of the 8×8 board and has only two types of pieces (man and king). In addition, the rules of Checkers, in which pieces can only move forwards and captures are forced, keep the tree from branching too quickly. For Chess, even a weak solution seems out of reach for the foreseeable future.

Chapter 2

The Basics

> An algorithm is a finite answer to an infinite number of questions.
>
> Stephen Kleene

As we saw in the Prologue, there seems to be some mysterious difference between Eulerian and Hamiltonian paths that makes one of them much harder to find than the other. To put this differently, Hamiltonianness seems to be a qualitatively more subtle property than Eulerianness. Why is one of these problems easy, while the other is like searching for a needle in a haystack?

If we want to understand the nature of these problems, we need to go beyond particular puzzles like the bridges of Königsberg or the edges of the dodecahedron. We need to ask how hard these problems are in general—for cities with any number of bridges, or graphs of any size—and ask how their complexity grows with the size of the city or the graph. To a computer scientist, the "complexity" of a problem is characterized by the amount of computational resources required to solve it, such as how much time or memory we need, and how these requirements grow as the problem gets larger.

In order to measure the complexity of a problem, we will think about the best possible *algorithm*, or computer program, that solves it. However, as we will see, computational complexity theory is not about how to write better programs, any more than physics is about building better spaceships. It is about understanding the underlying structure of different problems, and asking fundamental questions about them—for instance, whether they can be broken into smaller pieces that can be solved independently of each other.

2.1 Problems and Solutions

Let's start this chapter by saying precisely what we mean by a "problem," and what constitutes a "solution." If you have never thought about computational complexity before, our definitions may seem slightly counterintuitive. But as we will see, they give us a framework in which we can clearly state, and begin to answer, the questions posed in the Prologue.

2.1.1 What's the Problem?

Any particular instance of a problem, such as the Königsberg bridges, is just a finite puzzle. Once we have solved it, there is no more computation to do. On the other hand, Euler's generalization of this puzzle,

EULERIAN PATH
Input: A graph G
Question: Does there exist an Eulerian path on G?

is a worthy object of study in computational complexity theory—and we honor it as such by writing its name in elegant small capitals. We can think of this as an infinite family of problems, one for each graph G. Alternately, we can think of it as a function that takes a graph as its input, and returns the output "yes" or "no."

To drive this point home, let's consider a somewhat comical example. How computationally complex is Chess? Well, if you mean the standard game played on an 8×8 board, hardly at all. There are only a finite number of possible positions, so we can write a book describing the best possible move in every situation. This book will be somewhat ungainly—it has about 10^{50} pages, making it difficult to fit on the shelf—but once it is written, there is nothing left to do.

2.1

Now that we have disposed of Chess, let's consider a more interesting problem:

GENERALIZED CHESS
Input: A position on an $n \times n$ board, with an arbitrary number of pieces
Question: Does White have a winning strategy?

Now you're talking! By generalizing to boards of any size, and generalizing the rules appropriately, we have made it impossible for any finite book, no matter how large, to contain a complete solution. To solve this problem, we have to be able to solve Chess problems, not just look things up in books. Moreover, generalizing the problem in this way allows us to consider how quickly the game tree grows, and how much time it takes to explore it, as a function of the board size n.

Another important fact to note is that, when we define a problem, we need to be precise about what input we are given, and what question we are being asked. From this point of view, cities, graphs, games, and so on are neither complex nor simple—specific questions about them are.

These questions are often closely related. For instance, yes-or-no questions like whether or not a Hamiltonian cycle exists are called *decision problems*, while we call the problem of actually finding such a cycle a *search problem* or *function problem*. Problem 1.10 showed that if we can solve the decision version of HAMILTONIAN CYCLE, then we can also solve the search version. But there are also cases where it is easy to show that something exists, but hard to actually find it.

2.1.2 Solutions and Algorithms

Now that we've defined what we mean by a problem, what do we mean by a solution? Since there are an infinite number of possible graphs G, a solution to EULERIAN PATH can't consist of a finite list of answers we can just look up. We need a general method, or *algorithm*, which takes a graph as input and returns the correct answer as output.

2.2

While the notion of an algorithm can be defined precisely, for the time being we will settle for an intuitive definition: namely, a series of elementary computation steps which, if carried out, will produce the desired output. For all intents and purposes, you can think of an algorithm as a computer program written in your favorite programming language: C++, JAVA, HASKELL, or even (ugh) FORTRAN. However, in order to talk about algorithms at a high level, we will express them in "pseudocode." This is a sort of informal programming language, which makes the flow of steps clear without worrying too much about the syntax.

As our first example, let us consider one of the oldest algorithms known. Given two integers a and b, we would like to know their greatest common divisor $\gcd(a,b)$. In particular, we would like to know if a and b are *mutually prime*, meaning that $\gcd(a,b)=1$.

We can solve this problem using Euclid's algorithm, which appears in his *Elements* and dates at least to 300 B.C. It relies on the following fact: d is a common divisor of a and b if and only if it is a common divisor of b and $a \bmod b$. Therefore,

$$\gcd(a,b) = \gcd(b, a \bmod b). \tag{2.1}$$

This gives us an algorithm in which we repeatedly replace the pair (a,b) with the pair $(b, a \bmod b)$. Since the numbers get smaller each time we do this, after a finite number of steps the second number of the pair will be zero. At that point, the gcd is equal to the current value of the first number, since 0 is divisible by anything.

2.3

We can express this as a *recursive* algorithm. When called upon to solve the problem $\gcd(a,b)$, it calls itself to solve the simpler subproblem $\gcd(b, a \bmod b)$, until it reaches the *base case* $b=0$ which is trivial to solve. Note that if $a<b$ in the original input, the first application of this function will switch their order and call $\gcd(b,a)$. Here is its pseudocode:

```
Euclid(a, b)
begin
    if b = 0 then return a;
    return Euclid(b, a mod b);
end
```

Calling this algorithm on the pair (120,33), for instance, gives

$$\begin{aligned} &\texttt{Euclid}(120,33) \\ =&\texttt{Euclid}(33,21) \\ =&\texttt{Euclid}(21,12) \\ =&\texttt{Euclid}(12,9) \\ =&\texttt{Euclid}(9,3) \\ =&\texttt{Euclid}(3,0) \\ =&3. \end{aligned}$$

Is this a good algorithm? Is it fast or slow? Before we discuss this question, we bring an important character to the stage.

2.1.3 Meet the Adversary

> The Creator determines and conceals the aim of the game, and it is never clear whether the purpose of the Adversary is to defeat or assist him in his unfathomable project... But he is concerned, it would seem, in preventing the development of any reasoned scheme in the game.
>
> H. G. Wells, *The Undying Fire*

Computer scientists live in a cruel world, in which a malicious adversary (see Figure 2.1) constructs instances that are as hard as possible, for whatever algorithm we are trying to use. You may have a lovely algorithm that works well in many cases, but beware! If there is any instance that causes it to fail, or to run for a very long time, you can rely on the adversary to find it.

The adversary is there to keep us honest, and force us to make ironclad promises about our algorithms' performance. If we want to promise, for instance, that an algorithm always succeeds within a certain amount of time, this promise holds in *every* case if and only if it holds in the *worst* case—no matter what instance the adversary throws at us.

This is not the only way to think about a problem's hardness. As we will see in Chapter 14, for some problems we can ask how well algorithms work on average, when the instance is chosen randomly rather than by an adversary. But for the most part, computational complexity theorists think of problems as represented by their worst cases. A problem is hard if there *exist* hard instances—it is easy only if *all* its instances are easy.

2.4 So, is finding the gcd more like EULERIAN PATH or HAMILTONIAN PATH? Euclid's algorithm stops after a finite number of steps, but how long does it take? We answer this question in the next section. But first, we describe how to ask it in the most meaningful possible way.

2.2 Time, Space, and Scaling

> It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one... We might, for instance, count the number of additions, subtractions, multiplications, divisions, recordings of numbers, and extractions of figures from tables.
>
> Alan M. Turing, 1947

Time and space—the running time of an algorithm and the amount of memory it uses—are two basic computational resources. Others include the number of times we evaluate a complicated function, the number of bits that two people need to send each other, or the number of coins we need to flip. For each of these resources, we can ask how the amount we need *scales* with the size of our problem.

2.2.1 From Physics to Computer Science

The notion of scaling is becoming increasingly important in many sciences. Let's look at a classic example from physics. In 1619, the German astronomer and mathematician Johannes Kepler formulated his "harmonic law" of planetary motion, known today as Kepler's third law. Each planet has a "year" or orbital

FIGURE 2.1: The adversary bringing us a really stinky instance.

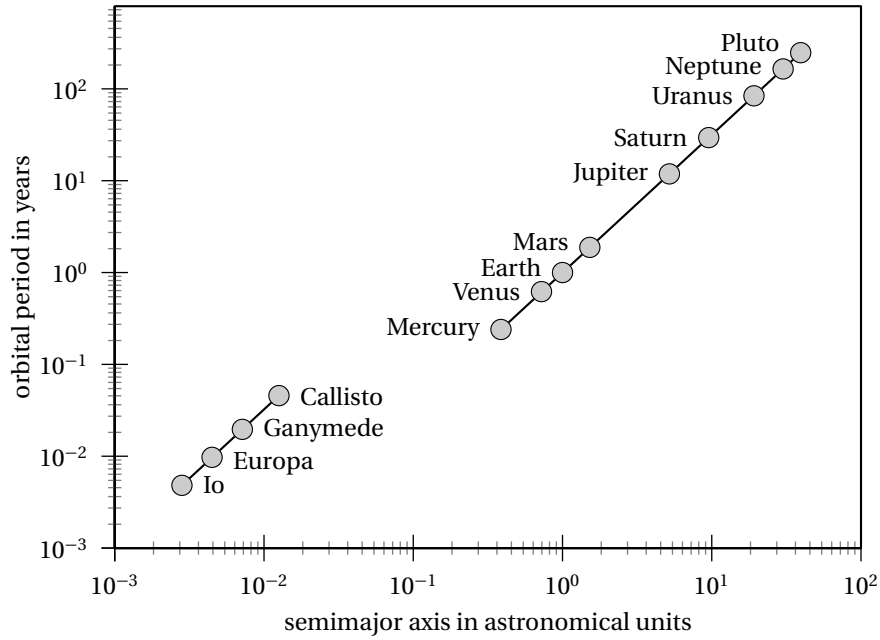


FIGURE 2.2: Scaling in physics: Kepler's harmonic law.

period T, and a distance R from the sun, defined as its semimajor axis since orbits are ellipses. Based on extensive observations, Kepler found that the ratio between T^2 and R^3 is the same for all planets in the solar system. We can write this as

$$T = C \cdot R^{3/2}, \tag{2.2}$$

for some constant C. If we plot T vs. R on a log–log plot as in Figure 2.2, this becomes

$$\log T = C' + \frac{3}{2} \log R,$$

where $C' = \log C$ is another constant. Thus all the planets—including Pluto, for tradition's sake—fall nicely on a straight line with slope 3/2.

What happens if we plot T vs. R for other systems, such as the four Galilean moons of Jupiter? The constant C in (2.2) depends on the central body's mass, so it varies from system to system. But the exponent 3/2, and therefore the slope on the log–log plot, remains the same. The scaling relationship (2.2) holds for any planetary system in the universe. It represents a fundamental property of celestial dynamics, which turns out to be Isaac Newton's celebrated law of gravitation. The constants don't matter—what matters is the way that T scales as a function of R.

In the same way, when we ask how the running time of an algorithm scales with the problem size n, we are not interested in whether your computer is twice as fast as mine. This changes the constant in front of the running time, but what we are interested in is how your running time, or mine, changes when n changes.

What do we mean by the "size" n of a problem? For problems like EULERIAN PATH, we can think of n as the number of vertices or edges in the input graph—for instance, the number of bridges or riverbanks in the city. For problems that involve integers, such as computing the greatest common divisor, n is the number of bits or digits it takes to express these integers. In general, the size of a problem instance is the amount of information I need to give you—say, the length of the email I would need to send—in order to describe it to you.

2.2.2 Euclidean Scaling

Let's look at a concrete example. How does the running time of Euclid's algorithm scale with n? Here n is the total number of digits of the inputs a and b. Ignoring the factor of 2, let's assume that both a and b are n-digit numbers.

The time it takes to calculate $a \bmod b$ depends on what method we use to divide a by b. Let's avoid this detail for now, and simply ask how many of these divisions we need, i.e., how many times Euclid's algorithm will call itself recursively before reaching the base case $b = 0$ and returning the result. The following exercise shows that we need at most a linear number of divisions.

Exercise 2.1 *Show that if $a \geq b$, then $a \bmod b < a/2$. Conclude from this that the number of divisions that Euclid's algorithm performs is at most $2\log_2 a$.*

If a has n digits, then $2\log_2 a \leq Cn$ for some constant C. Measuring n in bits instead of digits, or defining n as the total number of digits of a and b, just changes C to a different constant. But the number of divisions is always linear in n.

Following Kepler, we can check this linear scaling by making observations. In Figure 2.3 we plot the worst-case and average number of divisions as a function of n, and the relationship is clearly linear. If you are mathematically inclined, you may enjoy Problems 2.6, 2.7, and 2.8, where we calculate the slopes of these lines analytically. In particular, it turns out that the worst case occurs when a and b are successive Fibonacci numbers, which as Problem 2.4 discusses are well-known in the field of rabbit breeding.

Now that we know that the number of divisions in Euclid's algorithm is linear in n, what is its total running time? In a single step, a computer can perform arithmetic operations on integers of some fixed size, such as 64 bits. But if n is larger than this, the time it takes to divide one n-digit number by another grows with n. As Problem 2.11 shows, if we use the classic technique of long division, the total running time of Euclid's algorithm scales as n^2, growing polynomially as a function of n.

Let's dwell for a moment on the fact that the size n of a problem involving an integer a is the number of bits or digits of a, which is roughly $\log a$, rather than a itself. For instance, as the following exercise shows, the problem FACTORING would be easy if the size of the input were the number itself:

Exercise 2.2 *Write down an algorithm that finds the prime factorization of an integer a, whose running time is polynomial as a function of a (as opposed to the number of digits in a).*

However, we believe that FACTORING cannot be solved in an amount of time that is polynomial as a function of $n = \log a$, unless you have a quantum computer.

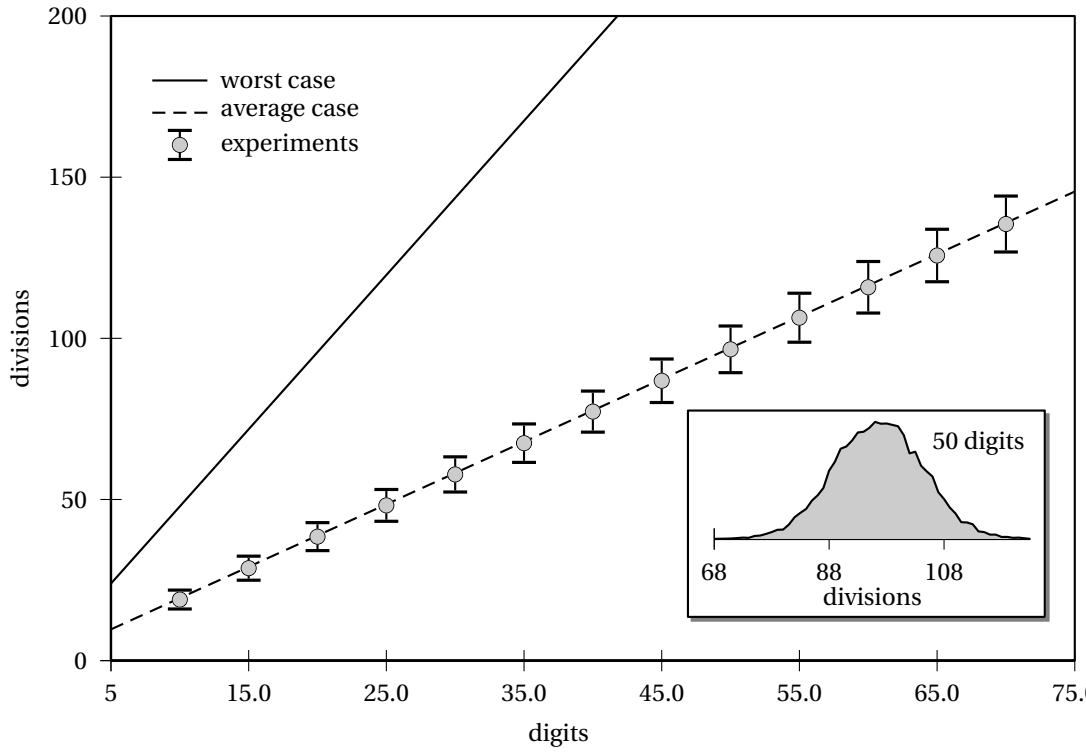


FIGURE 2.3: Scaling in computer science: the number of divisions done by Euclid's algorithm grows linearly with n when given n-digit inputs. In the inset, the typical distribution of a set of random inputs.

2.2.3 Asymptotics

In order to speak clearly about scaling, we will use asymptotic notation. This notation lets us focus on the qualitative growth of a function of n, and ignores the constant in front of it.

For instance, if the running time $f(n)$ of an algorithm grows quadratically, we say that $f(n) = \Theta(n^2)$. If it grows at most quadratically, but it might grow more slowly, we say that $f(n) = O(n^2)$. If it definitely grows less than quadratically, such as n^c for some $c < 2$, we say that $f(n) = o(n^2)$. Buying a faster computer reduces the constant hidden in Θ or O, but it doesn't change how $f(n)$ grows when n increases.

We define each of these symbols in terms of the limit as $n \to \infty$ of the ratio $f(n)/g(n)$ where $g(n)$ is whatever function we are comparing $f(n)$ with. Thus $f(n) = o(n^2)$ means that $\lim_{n\to\infty} f(n)/n^2 = 0$, while $f(n) = \Theta(n^2)$ means that $f(n)/n^2$ tends neither to zero nor to infinity—typically, that it converges to some constant C. We summarize all this in Table 2.1. In Appendix A.1 you can find formal definitions and exercises involving these symbols.

Exercise 2.3 *When we say that $f(n) = O(\log n)$, why don't we have to specify the base of the logarithm?*

These definitions focus on the behavior of $f(n)$ in the limit $n \to \infty$. This is for good reason. We don't really care how hard HAMILTONIAN PATH, say, is for small graphs. What matters to us is whether there is an

symbol	$C = \lim_{n\to\infty} \frac{f(n)}{g(n)}$	roughly speaking...
$f(n) = O(g(n))$	$C < \infty$	"$f \le g$"
$f(n) = \Omega(g(n))$	$C > 0$	"$f \ge g$"
$f(n) = \Theta(g(n))$	$0 < C < \infty$	"$f = g$"
$f(n) = o(g(n))$	$C = 0$	"$f < g$"
$f(n) = \omega(g(n))$	$C = \infty$	"$f > g$"

TABLE 2.1: A summary of asymptotic notation.

efficient algorithm that works for all n, and the difference between such an algorithm and an exhaustive search shows up when n is large. Even Kepler's law, which we can write as $T = \Theta(R^{3/2})$, only holds when R is large enough, since at small distances there are corrections due to the curvature of spacetime.

Armed with this notation, we can say that Euclid's algorithm performs $\Theta(n)$ divisions, and that its total running time is $O(n^2)$. Thus we can calculate the greatest common divisor of two n-digit numbers in polynomial time, i.e., in time $O(n^c)$ for a constant c. But what if Euclid's algorithm isn't the fastest method? What can we say about a problem's *intrinsic* complexity, as opposed to the running time of a particular algorithm? The next section will address this question with a problem, and an algorithm, that we all learned in grade school.

2.3 Intrinsic Complexity

For whichever resource we are interested in bounding—time, memory, and so on—we define the *intrinsic complexity* of a problem as the complexity of the *most efficient* algorithm that solves it. If we have an algorithm in hand, its existence provides an upper bound on the problem's complexity—but it is usually very hard to know whether we have the most efficient algorithm. One of the reasons computer science is such an exciting field is that, every once in a while, someone achieves an algorithmic breakthrough, and the intrinsic complexity of a problem turns out to be much less than we thought it was.

To illustrate this point, let's discuss the complexity of multiplying integers. Let $T(n)$ denote the time required to multiply two integers x and y which have n digits each. As Figure 2.4 shows, the algorithm we learned in grade school takes time $T(n) = \Theta(n^2)$, growing quadratically as a function of n.

This algorithm is so natural that it is hard to believe that one can do better, but in fact one can. One of the most classic ideas in algorithms is to *divide and conquer*—to break a problem into pieces, solve each piece recursively, and combine their answers to get the answer to the entire problem. In this case, we can break the n-digit integers x and y into pairs of $n/2$-digit integers as follows:

$$x = 10^{n/2}a + b \quad \text{and} \quad y = 10^{n/2}c + d\,.$$

Here a and b are the *high-order* and *low-order* parts of x, i.e., the first and second halves of x's digit sequence, and c and d are similarly the parts of y. Then

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd\,. \tag{2.3}$$

Of course, on a digital computer we would operate in binary instead of decimal, writing $x = 2^{n/2}a + b$ and so on, but the principle remains the same.

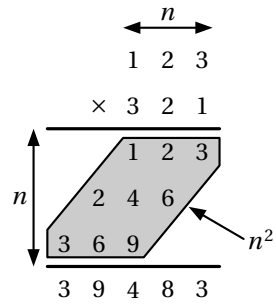


FIGURE 2.4: The grade-school algorithm for multiplication, which takes $\Theta(n^2)$ time to multiply n-digit integers.

This approach lets us reduce the problem of multiplying n-digit integers to that of multiplying several pairs of $n/2$-digit integers. We then reduce this problem to that of multiplying $n/4$-digit integers, and so on, until we get to integers so small that we can look up their products in a table. We assume for simplicity that n is even at each stage, but rounding $n/2$ up or down makes no difference when n is large.

What running time does this approach give us? If we use (2.3) as our strategy, we calculate four products, namely ac, ad, bc, and bd. Adding these products together is much easier, since the grade-school method of adding two n-digit integers takes just $O(n)$ time. Multiplying an integer by 10^n or $10^{n/2}$ is also easy, since all we have to do is shift its digits to the left and add n or $n/2$ zeros. The running time $T(n)$ then obeys the equation

$$T(n) = 4T(n/2) + O(n). \tag{2.4}$$

If $T(n)$ scales faster than linearly, then for large n we can ignore the $O(n)$ term. Then the running time is dominated by the four multiplications, and it essentially quadruples whenever n doubles. But as Problem 2.13 shows, this means that it grows quadratically, $T(n) = \Theta(n^2)$, just like the grade-school method. So we need another idea.

The key observation is that we don't actually need to do four multiplications. Specifically, we don't need ad and bc separately—we only need their sum. Now note that

$$(a+b)(c+d) - ac - bd = ad + bc. \tag{2.5}$$

Therefore, if we calculate $(a+b)(c+d)$ along with ac and bd, which we need anyway, we can obtain $ad+bc$ by subtraction, which like addition takes just $\Theta(n)$ time. Using this trick changes (2.4) to

$$T(n) = 3T(n/2) + O(n). \tag{2.6}$$

Now the running time only triples when n doubles, and using Problem 2.13 gives

$$T(n) = \Theta(n^\alpha) \text{ where } \alpha = \log_2 3 \approx 1.585.$$

So, we have tightened our upper bound on the complexity of multiplication from $O(n^2)$ to $O(n^{1.585})$.

Is this the best we can do? To be more precise, what is the smallest α for which we can multiply n-digit integers in $O(n^\alpha)$ time? It turns out that α can be arbitrarily close to 1. In other words, there are algorithms whose running time is less than $O(n^{1+\varepsilon})$ for any constant $\varepsilon > 0$. On the other hand, we have a lower bound of $T(n) = \Omega(n)$ for the trivial reason that it takes that long just to read the inputs x and y. These upper and lower bounds almost match, showing that the intrinsic complexity of multiplication is essentially linear in n. Thus multiplication turns out to be much less complex than the grade-school algorithm would suggest.


2.5

2.4 The Importance of Being Polynomial

> For practical purposes the difference between polynomial and exponential order is often more crucial than the difference between finite and non-finite.
>
> Jack Edmonds, 1965

Finding an algorithm that multiplies n-digit integers in $O(n^{1.585})$ time, instead of $O(n^2)$, reveals something about the complexity of multiplication. It is also of practical interest. If $n = 10^6$, for instance, this improves the running time by a factor of about 300 if the constants in the Os are the same.

However, the most basic distinction we will draw in computational complexity is between polynomial functions of n—that is, n raised to some constant—and exponential ones. In this section, we will discuss why this distinction is so important, and why it is so robust with respect to changes in our definition of computation.

2.4.1 Until the End of the World

One way to illustrate the difference between polynomials and exponentials is to think about how the size of the problems we can handle increases as our computing technology improves. Moore's Law (no relation) is the empirical observation that basic measures of computing technology, such as the density of transistors on a chip, are improving exponentially with the passage of time.


2.7

A common form of this law—although not Moore's original claim—is that processing speed doubles every two years. If the running time of my algorithm is $\Theta(n)$, doubling my speed also doubles the size n of problems I can solve in, say, one week. But if the running time grows as $\Theta(2^n)$, the doubling the speed just increases n by 1.

Thus whether the running time is polynomial or exponential makes an enormous difference in the size of problems we can solve, now and for the foreseeable future. We illustrate this in Figure 2.5, where we compare various polynomials with 2^n and $n!$. In the latter two cases, solving instances of size $n = 100$ or even $n = 20$ would take longer than the age of the universe. A running time that scales exponentially implies a harsh bound on the problems we can ever solve—even if our project deadline is as far away in the future as the Big Bang is in the past.

Exercise 2.4 *Suppose we can currently solve a problem of size n in a week. If the speed of our computer doubles every two years, what size problem will we be able to solve in a week four years from now, if the running time of our algorithm scales as $\log_2 n$, $\sqrt{n}$, n, n^2, 2^n, or 4^n?*

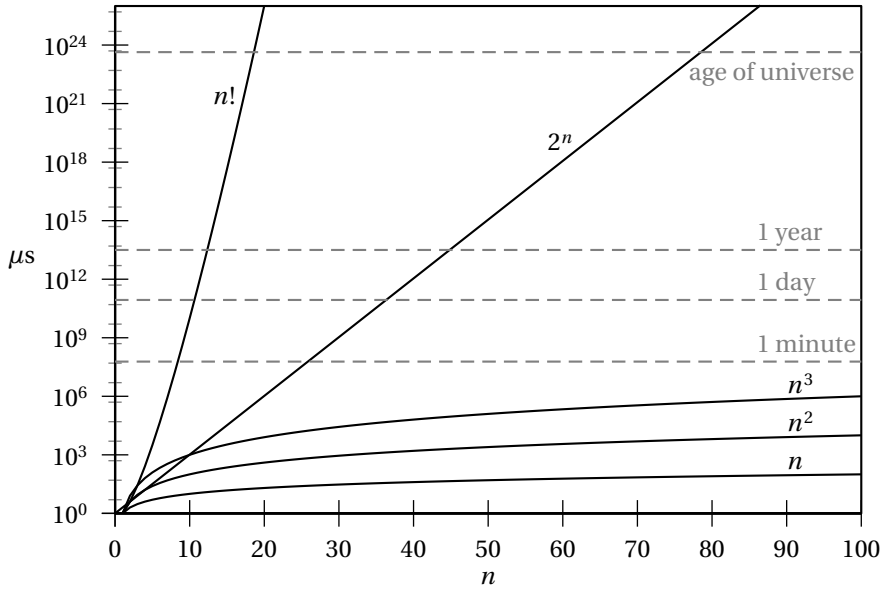


FIGURE 2.5: Running times of algorithms as a function of the size n. We assume that each one can solve an instance of size $n = 1$ in one microsecond. Note that the time axis is logarithmic.

```
Euler
input: a graph G = (V, E)
output: "yes" if G is Eulerian, and "no" otherwise
begin
    y := 0 ;
    for all v ∈ V do
        if deg(v) is odd then y := y + 1;
        if y > 2 then return "no";
    end
    return "yes"
end
```

FIGURE 2.6: Euler's algorithm for EULERIAN PATH. The variable y counts the number of odd-degree vertices.

2.4.2 Details, and Why they don't Matter

In the Prologue we saw that Euler's approach to EULERIAN PATH is much more efficient than exhaustive search. But how does the running time of the resulting algorithm scale with the size of the graph? It turns out that a precise answer to this question depends on many details. We will discuss just enough of these details to convince you that we can and should ignore them in our quest for a fundamental understanding of computational complexity.

In Figure 2.6 we translate Euler's Theorem into an algorithm, and express it in pseudocode. Quantifying the running time of this algorithm is not quite as trivial as it seems. To get us started, how many times will the **for** loop run? In the worst case, all vertices—or all but the last two—have even degree. Thus the **for** loop will, in general, run $O(|V|)$ times.

Next we need to specify how we measure the running time. The physical time, measured in seconds, will vary wildly from computer to computer. So instead, we measure time as the number of *elementary steps* that our algorithm performs. There is some ambiguity in what we consider "elementary," but let us assume for now that assignments like $y := 0$, arithmetical operations like $y + 1$, and evaluations of inequalities like $y > 2$ are all elementary and take constant time.

The next question is how long it takes to determine the degree of a vertex, which we denote $\deg(v)$. Clearly this is not very hard—we just have to count the number of neighbors v has. But analyzing it precisely depends on the format in which we are given the input graph G.

One common format is the *adjacency matrix*. This the $|V| \times |V|$ matrix A such that

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \text{ i.e., there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{otherwise.} \end{cases}$$

We could give this matrix directly as a $|V| \times |V|$ array of 0s and 1s. However, if G is a *sparse* graph that only has a few edges, then there are just a few pairs i, j such that $A_{ij} = 1$. As Problem 2.18 shows, we can then describe G more efficiently by giving a list of these pairs, or equivalently a list of edges.

Determining $\deg(v)$ takes different numbers of steps depending on which of these formats we use. Given the entire adjacency matrix, we would use a **for** loop with i ranging from 1 to $|V|$, and increment a counter each time $A_{vi} = 1$. Given a list of edges (i, j), we could scan the entire list and increment a counter each time either $i = v$ or $j = v$, and so on.

However, it is not even obvious that we can carry out instructions like "check if $A_{ij} = 1$" in a single elementary step. If the input is stored on a magnetic tape—an ancient memory technology which our reader is surely too young to remember—it might take a long time to roll the tape to the location of the data we wish to read. Among theoretical models of computation, a Turing machine, which we will discuss in Chapter 7, takes t steps to move to the tth location on its tape, while a machine with random access memory (RAM) can access any location in its memory in a single step. Thus moving from one of these models to another could change our running time considerably.

Finally, we need to agree how we specify the *size* of an instance. In general, this is the number n of bits it takes to describe it—if you like, the length of the email I would have to send you to tell you about it. This depends on our choice of input format, and n can be smaller or larger depending on whether this format is efficient or inefficient.

All these considerations make it difficult to quantify the running time precisely, and how it scales with the input size, without going into a great deal of detail about our input format, the particular implementation of our algorithm, and the type of machine on which we run our program. These are worthy engineering questions, but the goal of computational complexity theory is to take a larger view, and draw deep qualitative distinctions between problems. So, rather than studying the art of grinding lenses and mirrors, let us turn our attention to the stars.

2.4.3 Complexity Classes

As we saw in the previous section, one of the most basic questions we can ask about a problem is whether it can be solved in polynomial time as a function of its size. Let's consider the set of all problems with this property:

> P is the class of problems for which an algorithm exists that solves instances of size n in time $O(n^c)$ for some constant c.

Conversely, a problem is outside P if *no algorithm exists* that solves it in polynomial time—for instance, if the most efficient algorithm takes exponential time $2^{\varepsilon n}$ for some $\varepsilon > 0$.

P is our first example of a *complexity class*—a class of problems for which a certain kind of algorithm exists. We have defined it here so that it includes both decision problems, such as "does there exist an Eulerian path," and function problems, such as "construct an Eulerian path." Later on, many of our complexity classes will consist just of decision problems, which demand a yes-or-no answer.

More generally, for any function $f(n)$ we can define $\mathsf{TIME}(f(n))$ as follows:

> $\mathsf{TIME}(f(n))$ is the class of problems for which an algorithm exists that solves instances of size n in time $O(f(n))$.

In particular, P contains $\mathsf{TIME}(n)$, $\mathsf{TIME}(n^2)$, and so on, as well as noninteger exponents like $\mathsf{TIME}(n^{\log_2 3})$ which we met in Section 2.3. Formally,

$$\mathsf{P} = \bigcup_{c>0} \mathsf{TIME}(n^c).$$

The essential point is that we allow any exponent that is *constant* with respect to n. Exponents that grow as n grows, like $n^{\log n}$, are excluded from P. Throughout the book, we will use $\mathrm{poly}(n)$ as a shorthand for "$O(n^c)$ for some constant c," or equivalently for $n^{O(1)}$. In that case, we can write

$$\mathsf{P} = \mathsf{TIME}(\mathrm{poly}(n)).$$

2.8 If we wish to entertain running times that are exponentially large or even greater, we can define $\mathsf{EXP} = \mathsf{TIME}(2^{\mathrm{poly}(n)})$, $\mathsf{EXPEXP} = \mathsf{TIME}(2^{2^{\mathrm{poly}(n)}})$, and so on. This gives us a hierarchy of complexity classes, in which the amount of computation we can do becomes increasingly astronomical:

$$\mathsf{P} \subseteq \mathsf{EXP} \subseteq \mathsf{EXPEXP} \subseteq \cdots$$

But back down to earth. Why is the question of whether a problem can be solved in polynomial time or not so fundamental? The beauty of the definition of P is that it is extremely robust to changes in how we measure running time, and what model of computation we use. For instance, suppose we change our definition of "elementary step" so that we think of multiplying two integers as elementary. As long as they have only a polynomial number of digits, each multiplication takes polynomial time anyway, so this at most changes our running time from a polynomial to a smaller polynomial.

2.9 Similarly, going from a Turing machine to a RAM, or even a massively parallel computer—as long as it has only a polynomial number of processors—saves at most polynomial time. The one model of computation that seems to break this rule is a quantum computer, which we discuss in Chapter 15. So, to

be clear, we define P as the class of problems that *classical* computers, like the ones we have on our desks and in our laps today, can solve in polynomial time.

The class P is also robust with respect to most input formats. Any reasonable format for a graph, for example, has size n which is polynomial in the number of vertices, as long as it isn't a multigraph where some pairs of vertices have many edges between them. Therefore, we can say that a graph problem is in P if the running time is polynomial in $|V|$, and we will often simply identify n with $|V|$.

However, if we change our input format so drastically that n becomes exponentially larger or smaller, the computational complexity of a problem can change quite a bit. For instance, we will occasionally represent an integer a in *unary* instead of binary—that is, as a string of a ones. In that case, the size of the input is $n = a$ instead of $n = \log_2 a$. Exercise 2.2 shows that if we encode the input in this way, FACTORING can be solved in polynomial time.

Of course, this is just a consequence of the fact that we measure complexity as a function of the input size. If we make the input larger by encoding it inefficiently, the problem becomes "easier" in an artificial way. We will occasionally define problems with unary notation when we want some input parameter to be polynomial in n. But for the most part, if we want to understand the true complexity of a problem, it makes sense to provide the input as efficiently as possible.

Finally, P is robust with respect to most details of how we implement an algorithm. Using clever data structures, such as storing an ordered list in a binary tree instead of in an array, typically reduces the running time by a factor of n or n^2. This is an enormous practical improvement, but it still just changes one polynomial to another with a smaller power of n.

The fact that P remains unchanged even if we alter the details of our computer, our input format, or how we implement our algorithm, suggests that being in P is a *fundamental property of a problem*, rather than a property of how we humans go about solving it. In other words, the question of whether HAMILTONIAN PATH is in P or not is a mathematical question about the nature of Hamiltonian paths, not a subjective question about our own abilities to compute. There is no reason why computational complexity theory couldn't have been invented and studied thousands of years ago, and indeed there are glimmers of it here and there throughout history.

2.10

2.5 Tractability and Mathematical Insight

It is often said that P is the set of *tractable* problems, namely those which can be solved in a reasonable amount of time. While a running time of, say, $O(n^{10})$ is impractical for any interesting value of n, we encounter such large powers of n very rarely. The first theoretical results proving that a problem is in P sometimes give an algorithm of this sort, but within a few years these algorithms are usually improved to $O(n^3)$ or $O(n^4)$ at most.

Of course, even a running time of $O(n^3)$ is impractical if $n = 10^6$—for instance, if we are trying to analyze an online social network with 10^6 nodes. Now that fields like genomics and astrophysics collect vast amounts of data, stored on stacks of optical disks, containing far more information than your computer can hold at one time, some argue that even linear-time algorithms are too slow. This has given rise to a new field of *sublinear* algorithms, which examine only a small fraction of their input.

But for us, and for computational complexity theorists, P is not so much about tractability as it is about mathematical insight into a problem's structure. Both EULERIAN PATH and HAMILTONIAN PATH can be solved in exponential time by exhaustive search, but there is something different about EULERIAN PATH

that yields a polynomial-time algorithm. Similarly, when we learned in 2004 that the problem of telling whether an n-digit number is prime is in P, we gained a fundamental insight into the nature of PRIMALITY, even though the resulting algorithm (which we describe in Chapter 10) is not very practical.

The difference between polynomial and exponential time is one of kind, not of degree. When we ask whether a problem is in P or not, we are no longer just computer users who want to know whether we can finish a calculation in time to meet a deadline. We are theorists who seek a deep understanding of why some problems are qualitatively easier, or harder, than others.

2.11

Problems

> If there is a problem you can't solve, then there is an easier problem you can solve: find it.
>
> George Pólya, *How to Solve It*

2.1 Upgrades. The research lab of Prof. Flush is well-funded, and they regularly upgrade their equipment. Brilliant Pebble, a graduate student, has to run a rather large simulation. Given that the speed of her computer doubles every two years, if the running time of this simulation exceeds a certain T, she will actually graduate earlier if she waits for the next upgrade to start her program. What is T?

2.2 Euclid extended. Euclid's algorithm finds the greatest common divisor $\gcd(a,b)$ of integers a and b. Show that with a little extra bookkeeping it can also find (possibly negative) integers x and y such that

$$ax + by = \gcd(a,b). \tag{2.7}$$

Now assume that $b < a$ and that they are mutually prime. Show how to calculate the multiplicative inverse of b modulo a, i.e., the y such that $1 \le y < b$ and $by \equiv 1 \bmod a$.

Hint: the standard algorithm computes the remainder $r = a \bmod b$. The extended version also computes the quotient q in $a = qb + r$. Keep track of the quotients at the various levels of recursion.

2.3 Geometrical subtraction. Euclid's original algorithm calculated $a \bmod b$ by repeatedly subtracting b from a (by marking a line of length b off a line of length a) until the remainder is less than b. If a and b have n or fewer digits, show that this method can take exponential time as a function of n.

2.4 Fibonacci's rabbits. Suppose that I start my first year as a rabbit farmer with one baby rabbit. It takes a year for a baby rabbit to mature, and mature rabbits produce one baby per year. (Note that rabbits are immortal and reproduce asexually; elsewhere on the farm there are spherical cows.) If F_ℓ is the rabbit population in the ℓth year, show that the first few values of F_ℓ are $1, 1, 2, 3, 5, 8, \ldots$ and that in general, these obey the equation

$$F_\ell = F_{\ell-1} + F_{\ell-2}, \tag{2.8}$$

with the initial values $F_1 = F_2 = 1$. These are called the *Fibonacci numbers*. Show that they grow exponentially, as rabbits are known to do. Specifically, show that

$$F_\ell = \Theta(\varphi^\ell)$$

where φ is the "golden ratio,"

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618\ldots$$

In other words, find constants A and B for which you can prove by induction on ℓ that for all $\ell \geq 1$,

$$A\varphi^\ell \leq F_\ell \leq B\varphi^\ell .$$

Hint: φ is the largest root of the quadratic equation $\varphi^2 - \varphi - 1 = 0$. Equivalently, it is the unique positive number such that

$$\frac{\varphi}{1} = \frac{1+\varphi}{\varphi} .$$

2.5 Exponential growth, polynomial time. Using Problem 2.4, show that the problem of telling whether an n-digit number x is a Fibonacci number is in P. Hint: how many Fibonacci numbers are there between 1 and 10^n?

2.6 Euclid at his worst. Let's derive the worst-case running time of Euclid's algorithm. First, prove that the number of divisions is maximized when a and b are two adjacent Fibonacci numbers. Hint: using the fact that the smallest a such that $a \bmod b = c$ is $b + c$, work backwards from the base case and show that, if $b \leq a$ and finding $\gcd(a,b)$ takes ℓ divisions, then $a \geq F_{\ell+1}$ and $b \geq F_\ell$.

Now suppose that a and b each have n digits. Use Problem 2.4 to show that the number of divisions that Euclid's algorithm performs is at most $\log_\varphi a = C_{\text{worst}}\, n + O(1)$ where

$$C_{\text{worst}} = \frac{1}{\log_{10} \varphi} \approx 4.785 .$$

This is the slope of the upper line in Figure 2.3.

2.7 Euclid and Gauss. Now let's derive the average-case running time of Euclid's algorithm. First, let x denote the ratio b/a. Show that each step of the algorithm updates x as follows,

$$x = g(x) \text{ where } g(x) = \frac{1}{x} \bmod 1 . \tag{2.9}$$

Here by a number mod 1, we mean its fractional part. For instance, $\pi \bmod 1 = 0.14159...$ This function $g(x)$ is called the *Gauss map*, and its graph is shown in Figure 2.7. It also plays an important role in the theory of continued fractions: as we iterate (2.9), the integers $\lfloor 1/x \rfloor = (1/x) - (1/x \bmod 1)$ label the different "branches" of $g(x)$ we land on, and give the continued fraction series of x.

If we start with a random value of x and apply the Gauss map many times, we would expect x to be distributed according to some probability distribution $P(x)$. This distribution must be *stationary*: that is, it must remain the same when x is updated according to (2.9). This means that $P(x)$ must equal a sum of probabilities from all the values that x could have had on the previous step, adjusted to take the derivative of g into account:

$$P(x) = \sum_{y : g(y) = x} \frac{P(y)}{|g'(y)|} .$$

Show that one such distribution—which turns out to be essentially unique—is

$$P(x) = \frac{1}{\ln 2} \left(\frac{1}{x+1} \right) .$$

Of course, we haven't shown that the probability distribution of x converges to this stationary distribution. Proving this requires much more sophisticated arguments.

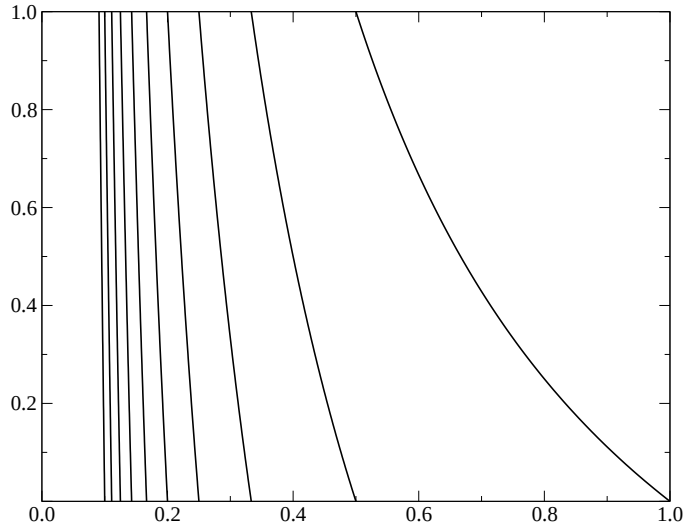


FIGURE 2.7: The Gauss map $g(x) = (1/x) \bmod 1$.

2.8 Euclid on average. Continuing from the previous problem, argue that the average number of divisions Euclid's algorithm does when given n-digit numbers grows as $C_{\text{avg}} n$ where

$$C_{\text{avg}} = -\frac{1}{\mathbb{E}[\log_{10} x]}\,.$$

Here $\mathbb{E}[\cdot]$ denotes the expectation given the distribution $P(x)$,

$$\mathbb{E}[\log_{10} x] = \int_0^1 P(x) \log_{10} x \, \mathrm{d}x\,.$$

Evaluate this integral to obtain

$$C_{\text{avg}} = \frac{12 \cdot \ln 2 \cdot \ln 10}{\pi^2} \approx 1.941\,.$$

This is the slope of the lower line in Figure 2.3. Clearly it fits the data very well.

2.9 The golden ratio again. To connect Problems 2.6 and 2.7, show that $1/\varphi = 0.618...$ is the largest fixed point of the Gauss map. In other words, it is the largest x such that $g(x) = x$. This corresponds to the fact that if a and b are successive Fibonacci numbers, the ratio $x = b/a$ stays roughly constant as Euclid's algorithm proceeds. Then show that, since $\lfloor \varphi \rfloor = 1$, the golden ratio's continued fraction expansion is

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$$

```
        3 1 4
    ---------
1 1 3 ) 3 5 5 0 0
        3 3 9
        -----
          1 6 0
          1 1 3
          -----
            4 7 0
            4 5 2
            -----
              1 8
```

FIGURE 2.8: Using long division to calculate a mod b. In this case $a = 35\,500$, $b = 113$, $\lfloor a/b \rfloor = 314$, and $a \bmod b = 18$.

Finally, show that cutting off this expansion after ℓ steps gives an approximation for φ as the ratio between two successive Fibonacci numbers,

$$\varphi \approx \frac{F_{\ell+1}}{F_\ell}.$$

2.10 Euclid and Fibonacci. Use Euclid's algorithm to show that any two successive Fibonacci numbers are mutually prime. Then, generalize this to the following beautiful formula:

$$\gcd(F_a, F_b) = F_{\gcd(a,b)}.$$

Note that we use the convention that $F_1 = F_2 = 1$, so $F_0 = 0$. Hint: you might want to look ahead at Problem 3.19.

2.11 Long division. Show that if a has n digits, $b \le a$, and the integer part $\lfloor a/b \rfloor$ of their ratio has m digits, then we can obtain a mod b in $O(nm)$ time. Hint: consider the example of long division shown in Figure 2.8, where $n = 5$ and $m = 3$. If you are too young to have been taught long division in school, humble yourself and ask your elders to teach you this ancient and beautiful art.

Now consider the series of divisions that Euclid's algorithm does, and show that the total time taken by these divisions is $O(n^2)$.

2.12 Divide and conquer. Show that for any constants $a, b > 0$, the recursive equation

$$T(n) = a\,T(n/b) \tag{2.10}$$

has the exact solution

$$T(n) = C\,n^{\log_b a},$$

where $C = T(1)$ is given by the base case.

2.13 Divide and a little more conquering. Now show that if $a > b > 1$ and we add a linear term to the right-hand side, giving

$$T(n) = a\,T(n/b) + C\,n,$$

then $T(n)$ is still $T(n) = \Theta(n^{\log_b a})$. In other words, prove by induction on n that there are constants A and B, depending on C and the initial conditions $T(1)$, such that

$$A\,n^{\log_b a} \le T(n) \le B\,n^{\log_b a}$$

for all $n \ge 1$. Feel free to assume that n is a power of b.

2.14 Toom's algorithm, part 1. After reading the divide-and-conquer algorithm for multiplying n-digit integers in Section 2.3, the reader might well ask whether dividing these integers into more than two pieces might yield an even better algorithm. Indeed it does!

To design a more general divide-and-conquer algorithm, let's begin by thinking of integers as polynomials. If x is an n-digit integer written in base b, and we wish to divide it into r pieces, we will think of it as a polynomial $P(z)$ of degree $r-1$, where $z=b^{n/r}$. For instance, if $x=314\,159\,265$ and $r=3$, then $x=P(10^3)$ where

$$P(z)=314z^2+159z+265.$$

Now, if $x=P(z)$ and $y=Q(z)$, their product xy is $R(z)$, where $R(z)=P(z)Q(z)$ is a polynomial of degree $2r-2$. In order to find R's coefficients, it suffices to sample it at $2r-1$ values of z, say at integers ranging from $-r+1$ to $r-1$. The coefficients of R are then linear combinations of these samples.

As a concrete example, suppose that $r=2$, and that $P(z)=az+b$ and $Q(z)=cz+d$. Then $R(z)=Az^2+Bz+C$ is quadratic, and we can find A, B, and C from three samples, $R(-1)$, $R(0)$ and $R(1)$. Write the 3×3 matrix that turns $(R(-1),R(0),R(+1))$ into (A,B,C), and show that the resulting algorithm is essentially identical to the one described in the text.

2.15 Toom's algorithm, part 2. Now let's generalize the approach of the previous problem to larger values of r. Each sample of $R(z)$ requires us to multiply two numbers, $P(z)$ and $Q(z)$, which have essentially n/r digits each. If we ignore the time it takes to multiply by M, the running time of this algorithm is

$$T(n)=(2r+1)\,T(n/r)$$

which by Problem 2.12 has the solution

$$T(n)=\Theta(n^\alpha) \text{ where } \alpha=\frac{\log 2r-1}{\log r}. \tag{2.11}$$

Show that α tends to 1 as r tends to infinity, so by taking r large enough we can achieve a running time of $O(n^{1+\varepsilon})$ for arbitrarily small ε.

2.16 Toom's algorithm, part 3. Let's continue from the previous problem. As r grows, the constant hidden in the Θ of (2.11) grows too, since we have to multiply the vector of samples by a matrix M of size $O(r^2)$. This suggests that the optimal value of r is some function of n. Show that, in the absence of any information about M's structure, a nearly optimal choice of r is

$$r=2^{\sqrt{\log n}}.$$

Then show that the running time of the resulting algorithm is

$$T(n)=n\,2^{O(\sqrt{\log n})}$$

which is less than $n^{1+\varepsilon}$ for any $\varepsilon>0$.

2.17 Fast matrix multiplication. Suppose we need to compute the product C of two $n\times n$ matrices A,B. Show that the naive algorithm for this takes $\theta(n^3)$ individual multiplications. However, we can do better, by again using a divide-and-conquer approach. Write

$$A=\begin{pmatrix}A_{1,1} & A_{1,2}\\ A_{2,1} & A_{2,2}\end{pmatrix},\ B=\begin{pmatrix}B_{1,1} & B_{1,2}\\ B_{2,1} & B_{2,2}\end{pmatrix},\text{ and } C=\begin{pmatrix}C_{1,1} & C_{1,2}\\ C_{2,1} & C_{2,2}\end{pmatrix},$$

where $A_{1,1}$ and so on are $n/2 \times n/2$ matrices. Now define the following seven $n/2 \times n/2$ matrices,

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}).
\end{aligned}$$

Then show that C is given by

$$\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Again using the fact that the cost of addition is negligible, show that this gives an algorithm whose running time is $\Theta(n^\alpha)$ where $\alpha = \log_2 7 \approx 2.807$. The optimal value of the exponent α is still not known.

2.6

2.18 How to mail a matrix. Given a graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, how many bits do we need to specify the adjacency matrix, and how many do we need to specify a list of edges? Keep in mind that it takes $\log n$ bits to specify an integer between 1 and n. When are each of these two formats preferable?

In particular, compare *sparse* graphs where $m = O(n)$ with *dense* graphs where $m = \Theta(n^2)$. How do things change if we consider a *multigraph*, like the graph of the Königsberg bridges, where there can be more than one edge between a pair of points?

2.19 We all live in a yellow subroutine. Another sense in which P is robust is that if one polynomial-time program uses another as a subroutine, then the total running time is still polynomial—at least if we're careful.

Suppose an algorithm B runs in polynomial time, and computes a function, e.g., a string or integer. Now suppose an algorithm A performs a polynomial number of steps, one of which calls B as a subroutine. Show that A also runs in polynomial time. Note that the input A sends to B might not be A's original input, but rather some other input that A is interested in. Hint: show that the set of polynomial functions is closed under composition. In other words, if $f(n)$ and $g(n)$ are both poly(n), so is their composition $f(g(n))$.

On the other hand, give an example where calling a polynomial-time algorithm B as a subroutine a polynomial number of times can give a total running time which is exponential. Hint: B's output might be bigger than its input. Can you think of other types of subroutine use where we can, or can't, guarantee polynomial time?

2.20 A little bit more than polynomial time. A *quasipolynomial* is a function of the form $f(n) = 2^{\Theta(\log^k n)}$ for some constant $k > 0$, where $\log^k n$ denotes $(\log n)^k$. Let us define QuasiP as the class of problems that can be solved in quasipolynomial time. First show that any quasipolynomial $f(n)$ with $k > 1$ is $\omega(g(n))$ for any polynomial function $g(n)$, and that $f(n) = o(h(n))$ for any exponential function $h(n) = 2^{\Theta(n^c)}$ for any $c > 0$. Thus $\mathsf{P} \subseteq \mathsf{QuasiP} \subseteq \mathsf{EXPTIME}$.

Then show that the set of quasipolynomial functions is closed under composition. Therefore, QuasiP programs can use each other as subroutines in the same sense that P programs can (see Problem 2.19) even if the main program gives an instance to the subroutine whose size is a quasipolynomial function of the original input size.

Notes

2.1 The Book of Chess. Schaeffer et al. [730] estimated that the number of legal positions in Checkers is 10^{18}. For Chess, the number of possible positions in a game 40 moves long was estimated at 10^{43} by Claude Shannon [746] in 1950. In 1994, Victor Allis [36] proved an upper bound of 5×10^{52} for the number of Chess positions and estimated the true number to be 10^{50}.

2.2 Dixit Algorizmi. The word *algorithm* goes back to the Persian Astronomer Muhammad ibn Musa al-Khwarizmi, born about 780 A.D. in Khwarezm (now Khiva in Uzbekistan). He worked in Baghdad, serving the caliph Abd-Allah al Mamun, son of the caliph Harun al-Rashid of *1001 Arabian Nights* fame. Al-Khwarizmi brought the Hindu number system to the Arab world, from where it spread to Europe, allowing us to write $31 \times 27 = 837$ instead of XXXI $\times$ XXVII $=$ DCCCXXXVII.

In medieval times, arithmetic was identified with al-Khwarizmi's name, and the formula *dixit Algorizmi* (thus spake al-Khwarizmi) was a hallmark of clarity and authority. Al-Khwarizmi's legacy is also found in the Spanish word *guarismo* (digit) and in the word *algebra*, which can be traced back to al-Khwarizmi's book on the solution of equations, the *Kitab al-muhtasar fi hisab al-gabr w'al-muqabalah*. A good reference on al-Khwarizmi, and the role of algorithms in Mathematics and Computer Science, is Knuth [493].

2.3 Euclid's algorithm. Euclid's algorithm appeared in his *Elements* in the 3rd century B.C. in Proposition 2 of Book VII. However, there is some evidence that it was known to Aristarchus and Archimedes [149]. The first proof that it finds the gcd of two n-digit numbers in $O(n)$ steps was given by Pierre-Joseph-Étienne Finck in 1841, who used the argument of Exercise 2.1 to get an upper bound of $2\log_2 a = (2\log_2 10)n$ on the number of divisions. This is probably the first nontrivial mathematical analysis of the running time of an algorithm. Three years later, Gabriel Lamé gave the bound of $5\log_{10} a = 5n$ using Fibonacci numbers, and the first part of Problem 2.6 is called Lamé's theorem. For a history of these results, see Shallit [741]. For a history of the average running time, discussed in Problems 2.7 and 2.8, see Knuth [495].

2.4 The adversary. Why should we focus on worst cases? Certainly other sciences, like physics, assume that problems are posed, not by a malicious adversary, but by Nature. Norbert Wiener draws a distinction between two kinds of devil, one that works cleverly against us, and another that simply represents our own ignorance [823, pp. 34–36]. Nature, we hope, is in the second category:

> The scientist is always working to discover the order and organization of the universe, and is thus playing a game against the arch-enemy, disorganization. Is this devil Manichaean or Augustinian? Is it a contrary force opposed to order or is it the very absence of order itself?
>
> ...This distinction between the passive resistance of nature and the active resistance of an opponent suggests a distinction between the research scientist and the warrior or the game player. The research physicist has all the time in the world to carry out his experiments, and he need not fear that nature will in time discover his tricks and method and change her policy. Therefore, his work is governed by his best moments, whereas a chess player cannot make one mistake without finding an alert adversary ready to take advantage of it and to defeat him. Thus the chess player is governed more by his worst moments than by his best moments.

There are actually three kinds of instance we might be interested in: worst-case ones, random ones, and real ones. In Problem 2.8 we derived the performance of Euclid's algorithm on random numbers, and in Chapter 14 we will consider problems based on random graphs and random Boolean formulas. But for many problems, there doesn't seem to be any natural way to define a random instance. Real-world problems are the most interesting to an engineer, but they typically have complicated structural properties that are difficult to capture mathematically. Perhaps the best way to study them is empirically, by going out and measuring them instead of trying to prove theorems.

Finally, one important reason why computer science focuses on the adversary is historical. Modern computer science got its start in the codebreaking efforts of World War II, when Alan Turing and his collaborators at Bletchley Park broke the Nazis' Enigma code. In cryptography, there really is an adversary, doing his or her best to break your codes and evade your algorithms.

2.5 Fast multiplication: Babbage, Gauss, and Fourier. The idea of multiplying two numbers recursively by dividing them into high- and low-order parts, and the fact that its running time is quadratic, was known to Charles Babbage—the 19th-century inventor of the Differential and Analytical Engines, whom we will meet in Chapter 7. He wanted to make sure that his Analytical Engine could handle numbers with any number of digits. He wrote [69, p. 125]:

> Thus if $a \cdot 10^{50} + b$ and $a' \cdot 10^{50} + b'$ are two numbers each of less than a hundred places of figures, then each can be expressed upon two columns of fifty figures, and a, b, a', b' are each less than fifty places of figures... The product of two such numbers is
>
> $$aa'10^{100} + (ab' + a'b)10^{50} + bb'.$$
>
> This expression contains four pairs of factors, $aa', ab', a'b, bb'$, each factor of which has less than fifty places of figures. Each multiplication can therefore be executed in the Engine. The time, however, of multiplying two numbers, each consisting of any number of digits between fifty and one hundred, will be nearly four times as long as that of two such numbers of less than fifty places of figures...
>
> Thus it appears that whatever may be the number of digits the Analytical Engine is capable of holding, if it is required to make all the computations with k times that number of digits, then it can be executed by the same Engine, but in an amount of time equal to k^2 times the former.

The trick of reducing the number of multiplications from four to three, and the resulting improvement in how the running time scales with the number of digits, is the sort of thing that Babbage would have loved. We will spend more time with Mr. Babbage in Chapter 7.

The first $O(n^{\log_2 3})$ algorithm for multiplying n-digit integers was found in 1962 by Karatsuba and Ofman [455]. However, the fact that we can reduce the number of multiplications from four to three goes back to Gauss! He noticed that in order to calculate the product of two complex numbers (where $\imath = \sqrt{-1}$)

$$(a + b\imath)(c + d\imath) = (ac - bd) + (ad + bc)\imath$$

we only need three real multiplications, such as ac, bd, and $(a + c)(b + d)$, since we can get the real and imaginary parts by adding and subtracting these products. The idea of [455] is then to replace $\imath$ with $10^{n/2}$, and to apply this trick recursively.

Toom [794] recognized that we can think of multiplication as interpolating a product of polynomials as described in Problems 2.14–2.16, and thus achieved a running time of $O(n^{1+\varepsilon})$ for arbitrarily small ε. This is generally called the Toom–Cook algorithm, since Stephen Cook also studied it in his Ph.D. thesis.

In 1971, Schönhage and Strassen [733] gave an $O(n \cdot \log n \cdot \log\log n)$ algorithm. The idea is to think of an integer x as a function, where $x(i)$ is its ith digit. Then, except for carrying, the product xy is the convolution of the corresponding functions, and the Fourier transform of their convolution is the product of their Fourier transforms. They then use the Fast Fourier Transform algorithm, which as we will discuss in Section 3.2.3 takes $O(n \log n)$ time. We outline this algorithm in Problems 3.15 and 3.16; we can also think of it as a special case of the Toom–Cook algorithm, where we sample the product of the two polynomials at the $2r$th roots of unity in the complex plane. An excellent description of this algorithm can be found in Dasgupta, Papadimitriou and Vazirani [215].

In 2007, Fürer [306] improved this algorithm still further, obtaining a running time of $n \cdot \log n \cdot 2^{O(\log^* n)}$. Here $\log^* n$ is the number of times we need to iterate the logarithm to bring n below 2; for instance, $\log^* 65536 = 4$ and $\log^* 10^{10000} < 5$. Since $\log^* n$ is "nearly constant," it seems likely that the true complexity of multiplication is $\Theta(n \log n)$. And in fact, this has been announced by Havey and Van Der Hoeven at the time we write this [374].

2.6 Matrix multiplication. The problem of calculating *matrix* products also has a long and interesting history. Multiplying two $n \times n$ matrices requires n^3 multiplications if we use the textbook method, but algorithms that work in time $O(n^\alpha)$ have been achieved for various $\alpha < 3$. In 1969, Strassen obtained the algorithm of Problem 2.17, for which $\alpha = \log_2 7 \approx 2.807$. Coppersmith and Winograd [202] presented an algorithm with $\alpha \approx 2.376$. Their algorithm has recently been improved [826, 219, 522] and as of 2017, the fastest variant has $\alpha \approx 2.3728639$.

While clearly $\alpha \geq 2$ since we need $\Omega(n^2)$ time just to read the input, it is not known what the optimal value of α is. However, there is some very promising recent work on algebraic approaches by Cohn and Umans [190] and Cohn, Kleinberg, Szegedy and Umans [188]. These include reasonable conjectures which would imply that $\alpha = 2$, or more precisely, that we can multiply matrices in time $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

2.7 Moore's Law. Gordon Moore, a co-founder of Intel, originally claimed in 1965 that the number of transistors in an integrated circuit roughly doubled each year. He later changed the doubling time to two years, and "Moore's Law" came to mean a similar claim about speed, memory per dollar, and so on. While clock speeds have recently leveled off, the real speed of computation measured in instructions per second continues to rise due to improvements in our computers' architecture, such has having multiple processors on a single chip, speeding up memory access by cleverly predicting what data the program will need next, and so on.

It could be argued that, at this point, Moore's Law has become a self-fulfilling prophecy driven by consumer expectations—now that we are used to seeing multiplicative improvements in our computers every few years, this is what we demand when we buy new ones.

Some technologists also believe that improvements in computing technology are even better described by Wright's Law. This states that as manufacturers and engineers gain more experience, technology improves polynomially as a function of the number of units produced. In this case, the exponential improvements we see are due to the exponential growth in the number of computers produced so far.

However, these improvements cannot continue forever without running up against fundamental physical constraints. If the current exponential growth in chip density continues, by around 2015 or 2020 our computers will use one elementary particle for each bit of memory. At these scales, we cannot avoid dealing with quantum effects, such as electron "tunneling" through potential barriers and jumping from one location to another. These effects will either be a source of noise and inconvenience—or, as we discuss in Chapter 15, of new computational power.

2.8 Exponentials. Some readers may find it jarring that functions of the form 2^{n^c} for any constant c are called simply "exponential." However, allowing the exponent to be polynomial in n, rather than simply linear, gives the class EXP the same robustness to the input format that P possesses.

2.9 Elementary steps. We need to be somewhat careful about how liberally we define the notion of an "elementary step." For instance, Schönhage [732] showed that machines that can multiply and divide integers of arbitrary size in a single step can solve PSPACE-complete problems in polynomial time (we will meet the class PSPACE in Chapter 8). Hartmanis and Simon [371] found that the same is true of machines that can perform bitwise operations on strings of arbitrary length in a single step. Finally, Bertoni, Mauri, and Sabadini [108] showed that such machines can solve #P-complete problems, which we will meet in Chapter 13. A review can be found in van Emde Boas [806].

Both PSPACE and #P are far above P in the complexity hierarchy. Thus if we assume that arithmetic operations can be performed in constant time regardless of the size of the numbers involved, we lose our ability to draw distinctions between hard problems and easy ones. We get a more meaningful picture of complexity if we assume that the cost of arithmetic operations is logarithmic in the size and accuracy of the numbers, i.e., linear in the number of digits or bits that are being manipulated. And this assumption is certainly more realistic, at least where digital computers are concerned.

2.10 The history of polynomial time. Computational complexity theory as we know it began with the 1965 paper of Juris Hartmanis and Richard Stearns [372], for which they received the Turing Award in 1993. Their paper defines

classes of functions by how much time it takes to compute them, proves by diagonalization (as we will discuss in Chapter 6) that increasing the computation time yields an infinite hierarchy of more and more powerful classes, and notes that changing the type of machine can alter the running time from, say, $\Theta(n)$ to $\Theta(n^2)$. They also make what has turned out to be a rather impressive understatement:

> It is our conviction that numbers and functions have an intrinsic computational nature according to which they can be classified... and that there is a good opportunity here for further research.

At around the same time, the idea that polynomial time represents a good definition of tractable computation appeared in the work of Cobham [181] and Edmonds [260]. Cobham says:

> The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? ... There seems to be no substantial problem in showing that using the standard algorithm it is in general harder—in the sense that it takes more time or more scratch paper—to multiply two decimal numbers than to add them. But this does not answer the question, which is concerned with ... properties intrinsic to the functions themselves and not with properties of particular related algorithms.

He goes on to define a class $\mathscr{L}$ of functions that can be computed in polynomial time as a function of the number of digits of their input, and recognizes that changing from one type of machine to another typically changes the power of the polynomial but preserves $\mathscr{L}$ overall.

Edmonds studied a polynomial-time algorithm for MAX MATCHING, an optimization problem that asks how to form as many partnerships as possible between neighboring vertices in a graph. He presents his result by calling it a *good* algorithm, and says:

> There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically [i.e., polynomially] with the size of the graph. The mathematical significance of this paper rests largely on the assumption that the two preceding sentences have mathematical meaning...

He then proposes that any algorithm can be broken down into a series of elementary steps, and that once we agree on what types of steps are allowed, the question of whether an algorithm exists with a given running time becomes mathematically well-defined.

For an even earlier discussion of whether mathematical proofs can be found in polynomial time, see the 1956 letter of Gödel to von Neumann discussed in Section 6.1.

2.11 The Robertson–Seymour Theorem. There are strange circumstances in which we can know that a problem is in P, while knowing essentially nothing about how to solve it. To see how this could be the case, let's start with a simple graph property.

A graph is *planar* if it can be drawn in the plane without any edges crossing each other. Kuratowski [511] and Wagner [816] showed that G is planar if and only if it does not contain either of the graphs K_5 or $K_{3,3}$ shown in Figure 2.9 as a *minor*, where a minor is a graph we can obtain from G by removing vertices or edges, or by shrinking edges and merging their endpoints. With some work, we can check for both these minors in polynomial time. While this is far from the most efficient algorithm, it shows that PLANARITY is in P.

Planarity is an example of a *minor-closed* property. That is, if G is planar then so are all its minors. Other examples of minor-closed properties include whether G can be drawn on a torus with no edge crossings, or whether it can be embedded in three-dimensional space in such a way that none of its cycles are knotted, or that no two cycles are linked. For any fixed k, the property that G has a vertex cover of size k or less (see Section 4.2.4) is also minor-closed.

Wagner conjectured that for every minor-closed property, there is a finite list $\{K_1, K_2, \ldots\}$ of excluded minors such that G has that property if and only if it does not contain any of them. After a series of 20 papers, Neil Robertson and

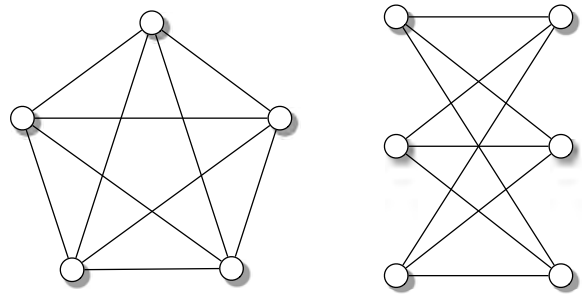


FIGURE 2.9: A graph is planar if and only if it does not contain either of these graphs as a minor.

Paul Seymour proved this conjecture in 2004 [705]. Along the way, they proved that for any fixed K, we can check in $O(n^3)$ time whether a graph with n vertices contains K as a minor.

As a result, we know that for any minor-closed property, the problem of telling whether a graph has it or not is in P. But Robertson and Seymour's proof is *nonconstructive*: it tells us nothing about these excluded minors, or how big they are. Moreover, while their algorithm runs in $O(n^3)$ time, the constant hidden in O depends in a truly horrendous way on the number of vertices in the excluded minors K_i (see [441] for a review). We are thus in the odd position of knowing that an entire family of problems is in P, without knowing polynomial-time algorithms for them, or how long they will take.

Chapter 3

Insights and Algorithms

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, i.e., express it as an algorithm... The attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.
>
> Donald E. Knuth

To the townspeople of Königsberg, the set of possible paths across the seven bridges seemed like a vast, formless mist. How could they find a path, or tell whether one exists, without an arduous search? Euler's insight parted this mist, and let them see straight to the heart of the problem.

Moreover, his insight is not just a sterile mathematical fact. It is a living, breathing algorithm, which solves the Bridges of Königsberg, or even of Venice, quickly and easily. As far as we know, whether a problem is in P or not depends on whether an analogous insight exists for it: some way we can guide our search, so that we are not doomed to wander in the space of all possible solutions.

But while mathematical insights come in many forms, we know of just a few major strategies for constructing polynomial-time algorithms. These include *divide and conquer*, where we break problems into easier subproblems; *dynamic programming*, where we save time by remembering subproblems we solved before; *greedy algorithms*, which start with a bad solution or none at all, and make small changes until it becomes the best possible one; *duality*, where two seemingly different optimization problems turn out to have the same solution, even though they approach it from opposite directions; and *reductions*, which transform one problem into another that we already know how to solve.

Why do these strategies work for some problems but not for others? How can we break a problem into subproblems that are small enough, and few enough, to solve quickly? If we start with a bad solution, can we easily feel our way towards a better one? When is one problem really just another one in disguise? This chapter explores these questions, and helps us understand why some problems are easier than they first appear. Along the way, we will see how to sort a pack of cards, hear the music of the spheres, typeset beautiful books, align genomes, find short paths, build efficient networks, route the flow of traffic, and run a dating service.

3.1

3.1 Recursion

We have already seen two examples of recursion in Chapter 2: Euclid's algorithm for the greatest common divisor, and the divide-and-conquer algorithm for multiplying n-digit integers. These algorithms work by creating "children"—new incarnations of themselves—and asking them to solve smaller versions of the same problem. These children create their own children in turn, asking them to solve even smaller problems, until we reach a base case where the problem is trivial.

We start this chapter with another classic example of recursion: the Towers of Hanoi, introduced by the mathematician Edouard Lucas under the pseudonym of "N. Claus de Siam." While this is really just a puzzle, and not a "problem" in the sense we defined in Chapter 2, it is still an instructive case of how a problem can be broken into subproblems. The story goes like this:

> In the great temple at Benares, beneath the dome that marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate, and the others getting smaller and smaller up to the top one...
>
> Day and night, unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When the sixty-four disks shall have been thus transferred from the needle which at creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust and with a thunderclap the world will vanish.

This appears to be a product of the French colonial imagination, with Hanoi and Benares chosen as suitably exotic locations. Presumably, if a Vietnamese mathematician had invented the puzzle, it would be called the Towers of Eiffel.

A little reflection reveals that one way to move all n disks from the first peg to the second is to first move $n-1$ disks to the third peg, then move the largest disk from the first to the second, and then move the $n-1$ disks from the third peg to the second. But how do we move these $n-1$ disks? Using exactly the same method. This gives the algorithm shown in Figure 3.1.

We can think of running a recursive algorithm as traversing a tree. The root corresponds to the original problem, each child node corresponds to a subproblem, and the individual moves correspond to the leaves. Figure 3.2 shows the tree for $n=3$, with the solution running from top to bottom.

If the number of disks is n, what is the total number of moves we need? Let's denote this $f(n)$. Since our algorithm solves this problem twice for $n-1$ disks and makes one additional move, $f(n)$ obeys the equation

$$f(n)=2f(n-1)+1\,. \tag{3.1}$$

The base case is $f(0)=0$, since it takes zero moves to move zero disks. The solution is given by the following exercise.

Exercise 3.1 *Prove by induction on n that the solution to* (3.1) *with the base case $f(0)=0$ is*

$$f(n)=2^n-1\,.$$

```
Hanoi(n, i, j) // move n disks from peg i to peg j
begin
    if n = 0 then return;
    Hanoi(n − 1, i, k);
    move a disk from peg i to peg j ;
    Hanoi(n − 1, k, j);
end
```

FIGURE 3.1: The recursive algorithm for solving the Towers of Hanoi. Here k denotes the third peg, other than i and j. Note that i, j, and k are "local variables," whose values change from one incarnation of the algorithm to the next. Note also that in the base case $n = 0$, the algorithm simply returns, since there is nothing to be done.

In fact, as Problem 3.1 asks you to show, this algorithm is the best possible, and $2^n - 1$ is the smallest possible number of moves. Thus the priests in the story need to perform $2^{64} - 1 \approx 1.8 \times 10^{19}$ moves, and it seems that our existence is secure for now. If the number of moves were only, say, 9 billion, we might be in trouble if the priests gain access to modern computing machinery.

3.2

3.2 Divide and Conquer

Like the solution to the Towers of Hanoi, many recursive algorithms work by breaking a problem into several pieces, finding the answer to each piece, and then combining these answers to obtain the answer to the entire problem. We saw this in Section 2.3, where we multiplied n-digit numbers by breaking them into pairs of $n/2$-digit numbers.

Perhaps the simplest example of this approach is *binary search*. If I want to look up a word w in a dictionary, I can compare w to the word in the middle, and then focus my search on the first or second half of the dictionary. Using the same approach lets me focus on one-quarter of the dictionary, and so on. Since each step divides the dictionary in half, I can find w in a dictionary of N words in just $\log_2 N$ steps.

In this section, we will see several important problems where a divide-and-conquer strategy works. These include sorting large lists, raising numbers to high powers, and finding the Fourier transform of an audio signal. In each case, we solve a problem recursively by breaking it into independent subproblems, solving each one, and then combining their results in some way.

3.2.1 Set This House in Order: Sorting

> We start with the smallest. Then what do we do?
> We line them all up. Back to back. Two by two.
> Taller and taller. And, when we are through,
> We finally will find one who's taller than who.
>
> Dr. Seuss, *Happy Birthday To You!*

Suppose I wish to sort a pack of cards using the divide-and-conquer strategy. I start by splitting the pack into two halves, and sorting each one separately. I then merge the two sorted halves together, so that the

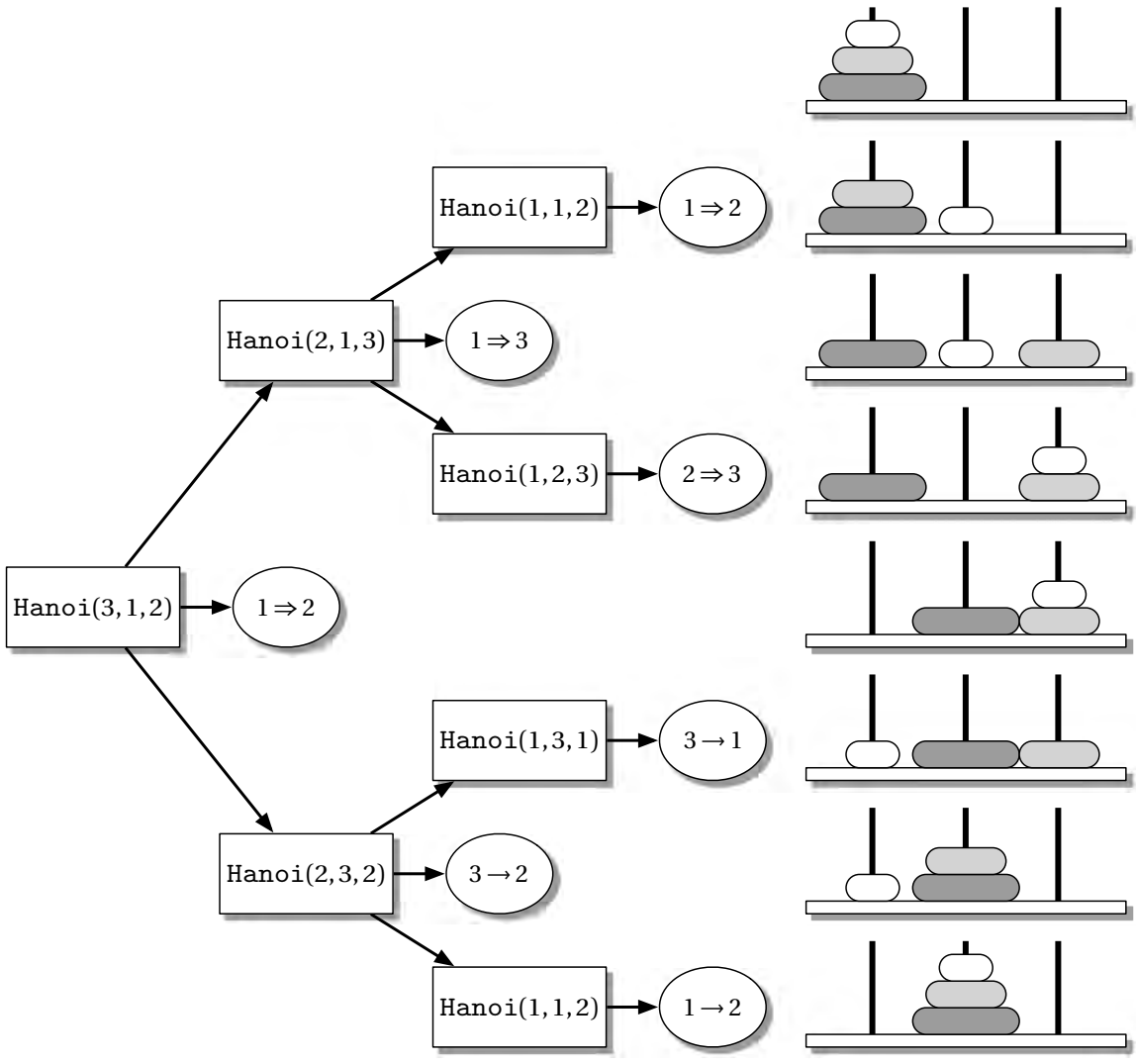


FIGURE 3.2: The tree corresponding to the recursive algorithm for the Towers of Hanoi with $n = 3$. Each node `Hanoi`(n, i, j) corresponds to the subproblem of moving n disks from peg i to peg j. The root node, corresponding to the original problem, is at the left. The actual moves appear on the leaf nodes (the ellipses), and the solution goes from top to bottom.

```
Mergesort(ℓ)
input: a list ℓ of n elements
output: a sorted version of ℓ
begin
    if |ℓ| ≤ 1 then return;
    ℓ1 := the first half of ℓ ;
    ℓ2 := the second half of ℓ ;
    ℓ1 := Mergesort(ℓ1);
    ℓ2 := Mergesort(ℓ2);
    return merge(ℓ1,ℓ2);
end
```

FIGURE 3.3: The `Mergesort` algorithm.

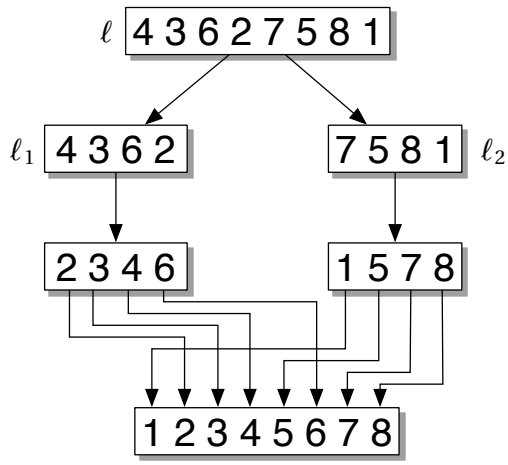


FIGURE 3.4: `Mergesort` splits the list ℓ into two halves, and sorts each one recursively. It then merges the two sorted halves, taking elements from ℓ_1 or ℓ_2, whichever is smaller.

entire pack is sorted: think of a careful riffle shuffle, where I let a card fall from either my left hand or my right, depending on which of the two cards should come first. This gives a recursive algorithm shown in Figure 3.3, which we illustrate in Figure 3.4.

To quantify the running time of this algorithm, let's count the number of times it compares one element to another. Let $T(n)$ be the number of comparisons it takes to sort an element of length n. Assuming for simplicity that n is even, sorting the two halves of the list recursively takes $2T(n/2)$ comparisons. How many comparisons does the `merge` operation take? We start by comparing the elements at the heads of ℓ_1 and ℓ_2, moving whichever one is smaller to the final sorted list, and continuing until ℓ_1 or ℓ_2 is empty. This takes at most $n-1$ comparisons, but for simplicity we'll assume that it takes n. Then the total number

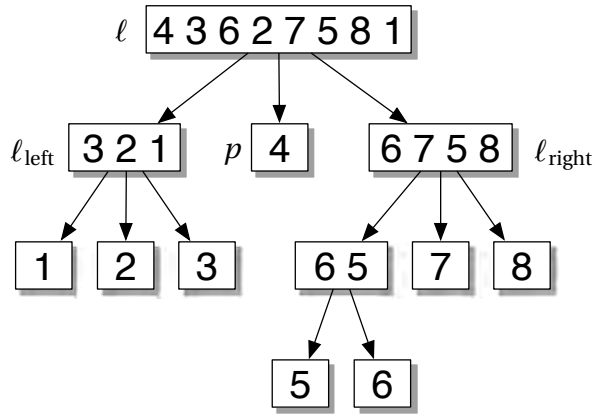


FIGURE 3.5: The recursive tree of Quicksort. At each step we choose a pivot p, and partition ℓ into sublists ℓ_{left} and ℓ_{right} depending on whether each element is smaller or larger than p. The leaves of the tree correspond to lists of size 1, for which no further sorting is necessary.

of comparisons is

$$T(n) = 2T(n/2) + n\,. \tag{3.2}$$

Since it takes zero comparisons to sort a list of size 1, the base case is $T(1) = 0$. As Problem 3.7 asks you to show, if we assume for simplicity that n is a power of two, the solution to (3.2) is

$$T(n) = n \log_2 n = \Theta(n \log n).$$

Now let's look at `Quicksort`. Like `Mergesort`, it is a divide-and-conquer algorithm, but now we break the list up in a different way. Instead of simply breaking it into halves, we choose a *pivot* element p, compare all the other elements to it, and put them in the left or right sublist according to whether they are smaller or larger than p. We then recursively sort these sublists as in Figure 3.5, subdividing the original list until we are left with lists of size 0 or 1. This gives the pseudocode shown in Figure 3.6.

`Quicksort` is interesting to us mainly in that it offers a contrast between worst-case and average-case running time. The number $T(n)$ of comparisons it takes to sort a list of size n depends on where the pivot p falls in the list. Let's say that p is the rth smallest element. Then there are $r-1$ elements smaller than p, and $n-r$ elements larger than p, and these elements end up in the left and right sublists respectively. Counting the $n-1$ comparisons it takes to compare p to everyone else, this gives the equation

$$T(n) = T(r-1) + T(n-r) + n - 1\,. \tag{3.3}$$

If we are very lucky and the pivot is always the median of the list, we have $r = n/2$. Ignoring the difference between n and $n-1$, this gives

$$T(n) = 2T(n/2) + n$$

just as we had for `Mergesort`, and $T(n) = n \log_2 n$.

```
Quicksort(ℓ)
input: a list ℓ of n elements
output: a sorted version of ℓ
begin
    if n ≤ 1 then return;
    choose a pivot p ∈ ℓ ;
    forall the x ∈ ℓ do
        if x < p then put x in ℓ_left ;
        else put x in ℓ_right ;
    end
    return {Quicksort(ℓ_left), p, Quicksort(ℓ_right)} ;
end
```

FIGURE 3.6: The `Quicksort` algorithm.

On the other hand, if we are very unlucky, the pivot is always the smallest or largest in the list, with $r = 1$ or $r = n$. In this case, we have succeeded only in whittling down a list of size n to one of size $n-1$, and the running time is

$$T(n) = T(n-1) + n - 1 .$$

With the base case $T(1) = 0$, this gives the arithmetic series

$$T(n) = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} = \Theta(n^2).$$

Since there is a large difference between the best case and the worst case, let's consider the *average* case, where r is uniformly random—that is, where it is equally likely to take any value between 1 and n. Averaging (3.3) over r gives us the following equation, where now $T(n)$ is the average number of comparisons:

$$\begin{aligned} T(n) &= n - 1 + \frac{1}{n}\sum_{r=1}^{n}\Big(T(r-1) + T(n-r)\Big) \\ &= n - 1 + \frac{2}{n}\sum_{r=1}^{n} T(r-1). \end{aligned} \tag{3.4}$$

While the average case is presumably not as good as the best case, we might still hope that $T(n)$ scales as $\Theta(n \log n)$, since most of the time the pivot is neither the largest nor the smallest element. Indeed, as Problem 3.9 shows—and, using another method, Problem 10.3—when n is large the solution to (3.4) is

$$T(n) \approx 2n \ln n .$$

Since

$$\frac{2n \ln n}{n \log_2 n} = 2 \ln 2 \approx 1.386 ,$$

the average number of comparisons is only 39% greater than it would be if the pivot were always precisely the median.

Now, why might r be uniformly random? There are two reasons why this could be. One is if we choose the pivot deterministically, say by using the first element as in Figure 3.5, but if the input list is in random order, where all $n!$ permutations are equally likely. This is all well and good, but in the world of computer science assuming that the input is random is overly optimistic. If our friend the adversary knows how we choose the pivot, he can give us an instance where the pivot will always be the smallest or largest element—in this case, a list that is already sorted, or which is sorted in reverse order. Thus he can saddle us with the worst-case running time of $\Theta(n^2)$.

However, the other reason r might be random is if we choose the pivot randomly, rather than deterministically. If p is chosen uniformly from the list then r is uniformly random, no matter what order the input is in. Instead of averaging over inputs, we average over the algorithm's choices, and achieve an average running time of $2n \ln n$ no matter what instance the adversary gives us. We will return to this idea in Chapter 10, where we explore the power of randomized algorithms.

Having seen both these algorithms, the reader might wonder whether it is possible to sort qualitatively faster than `Mergesort` or `Quicksort`. We will see in Section 6.2 that if we use comparisons, sorting n elements requires at least $\log_2 n! \approx n \log_2 n$ steps. Therefore, the number of comparisons that `Mergesort` performs is essentially optimal. This is one of the few cases where we can determine the optimal algorithm.

3.3

3.2.2 Higher Powers

> That sweet joy may arise from such contemplations cannot be denied. Numbers and lines have many charms, unseen by vulgar eyes, and only discovered to the unwearied and respectful sons of Art. In features the serpentine line (who starts not at the name) produces beauty and love; and in numbers, high powers, and humble roots, give soft delight.
>
> E. De Joncourt, as quoted by Charles Babbage

Let's look at another example of the divide-and-conquer strategy. Given x and y as inputs, how hard is it to calculate x^y?

If x and y each have n digits, x^y could be as large as 10^{n10^n} and have $n10^n$ digits. It would take an exponential amount of time just to write this number down, regardless of how long it takes to calculate it. So, in order to keep the result to at most n digits, we define the following problem:

MODULAR EXPONENTIATION
Input: n-digit integers x, y, and p
Output: $x^y \bmod p$

As we will see later, this problem is important in cryptography and algorithms for PRIMALITY. Is it in P?

An obvious approach is to start with $x^0 = 1$ and do y multiplications, increasing the power of x by one and taking the result mod p each time. But since y is exponentially large, this would take exponential time. A much better approach is to start with x, square it, square its square, and so on. This gives the powers

$$x, x^2, x^4, x^8, \ldots$$

```
Power(x, y, p)
input: integers x, y, p
output: x^y mod p
begin
    if y = 0 then return 1;
    t := Power(x, ⌊y/2⌋, p);
    if y is even then return t^2 mod p;
    else return x t^2 mod p;
end
```

FIGURE 3.7: The repeated-squaring algorithm for MODULAR EXPONENTIATION.

where we take the result mod p at each step. If y is a power of 2, we get x^y after just $\log_2 y = O(n)$ squarings. If y is not a power of 2, we can first derive x^{2^k} for all powers of 2 up to y, and then combine these according to y's binary digit sequence: for instance,

$$x^{999} = x \cdot x^2 \cdot x^4 \cdot x^{32} \cdot x^{64} \cdot x^{128} \cdot x^{256} \cdot x^{512}.$$

Since this product involves $O(n)$ powers, the total number of multiplications we need to do is still $O(n)$. Since we know how to multiply n-digit numbers in polynomial time, the total time we need is polynomial as a function of n. Therefore, MODULAR EXPONENTIATION is in P.

We can view this as a divide-and-conquer algorithm. Let $\lfloor y/2 \rfloor$ denote $y/2$ rounded down to the nearest integer. Then we calculate x^y recursively by calculating $x^{\lfloor y/2 \rfloor}$ and squaring it, with an extra factor of x thrown in if y is odd. This gives the algorithm shown in Figure 3.7.

The fact that we can get exponentially high powers by squaring repeatedly will come up several times in this book. For instance, by applying the same idea to matrix powers, we can find paths in graphs even when these graphs are exponentially large.

Modular exponentiation is also interesting because, as a function, it seems to be much harder to do backwards than forwards. Consider the following problem:

DISCRETE LOG
Input: n-digit integers x, z, and p
Output: An integer y, if there is one, such that $z = x^y \bmod p$

We call this problem DISCRETE LOG since we can think of y as $\log_x z$ in the world of integers mod p.

Our current belief is that, unlike MODULAR EXPONENTIATION, DISCRETE LOG is outside P. In other words, if we fix x and p, we believe that $f(y) = x^y \bmod p$ is a *one-way function*: a function in P, whose inverse is not. We will discuss pseudorandom numbers and cryptosystems based on this problem, and quantum algorithms that would break them, in Chapters 10 and 15.

3.2.3 The Fast Fourier Transform

We end this section with one more divide-and-conquer algorithm—one which is used today throughout digital signal processing, from speech recognition and crystallography to medical imaging and audio

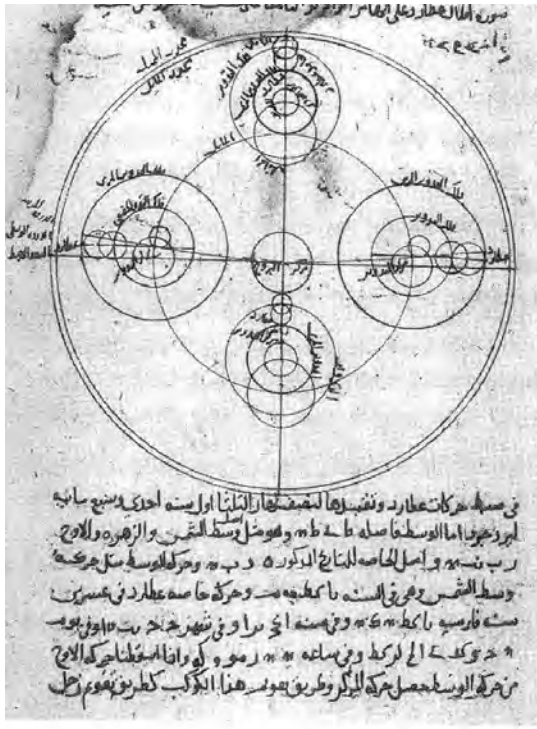


FIGURE 3.8: Fourier analysis through the ages. Left, Ibn al-Shāṭir's model of the motion of Mercury using six epicycles. Right, adjusting a coefficient in a tide-predicting machine.

compression. Readers who are unfamiliar with complex numbers should feel free to skip this section for now, but you'll need to understand it before we study quantum computing. First, some history.

Early Greek astronomers, much like modern physicists, were very fond of mathematical elegance. They regarded circles and spheres as the most perfect shapes, and postulated that the heavenly bodies move around the Earth in perfect circular orbits. Unfortunately, this theory doesn't fit the data very well. In particular, planets undergo *retrograde* motion, in which they move across the sky in the reverse of the usual direction.

To fix this problem while remaining faithful to the idea of circular motion, Ptolemy proposed that the planets move in *epicycles*, circles whose centers move around other circles. The position of each planet is thus a sum of two vectors, each of which rotates in time with a particular frequency. By adding more and more epicycles, we can fit the data better and better. By the 14th century, Islamic astronomers had produced Ptolemaic systems with as many as six epicycles (see Figure 3.8).

3.4

In a more terrestrial setting—but still with astronomical overtones—in 1876 Sir William Thomson, later Lord Kelvin, built a machine for predicting tides. It had a series of adjustable wheels, corresponding to combinations of the daily, lunar, and solar cycles. A system of pulleys, driven by turning a crank, summed these contributions and drew the resulting graph on a piece of paper. Tide-predicting machines like the one shown in Figure 3.8 were used as late as 1965.

The art of writing functions as a sum of oscillating terms is called Fourier analysis, in honor of Joseph Fourier, who studied it extensively in the early 19th century. We can do this using sums of sines and cosines, but a more elegant way is to use Euler's formula,

$$e^{\imath\theta} = \cos\theta + \imath \sin\theta\,.$$

Then we can write any smooth function $f(t)$ as

$$f(t) = \sum_{\alpha} \tilde{f}(\alpha) e^{\imath\alpha t}\,,$$

where the sum ranges over some set of frequencies α. The function $\tilde{f}$, which gives the coefficient for each α, is called the *Fourier transform* of f.

If rather than a continuous function, we have a discrete set of n samples $f(t)$ where $t = 0, 1, \ldots, n-1$, it suffices to consider frequencies that are multiples of $2\pi/n$. Let ω_n denote the nth root of 1 in the complex plane,

$$\omega_n = e^{2\imath\pi/n}\,.$$

Then this discrete set of frequencies $2\pi k/n$ gives us the *discrete Fourier transform*,

$$f(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \tilde{f}(k)\, \omega_n^{kt}\,. \tag{3.5}$$

The reason for the normalization factor $1/\sqrt{n}$ will become clear in a moment.

Now suppose we have a set of samples, and we want to find the Fourier transform. For instance, we have a series of observations of the tides, and we want to set the parameters of our tide-predicting machine. How can we invert the sum (3.5), and calculate $\tilde{f}(k)$ from $f(t)$?

The thing to notice is that if we think of f and $\tilde{f}$ as n-dimensional vectors, then (3.5) is just a matrix multiplication:

$$\begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \end{pmatrix} = \frac{1}{\sqrt{n}} \begin{pmatrix} 1 & 1 & 1 & \\ 1 & \omega_n & \omega_n^2 & \cdots \\ 1 & \omega_n^2 & \omega_n^4 & \\ & \vdots & & \ddots \end{pmatrix} \cdot \begin{pmatrix} \tilde{f}(0) \\ \tilde{f}(1) \\ \tilde{f}(2) \\ \vdots \end{pmatrix},$$

or

$$f = Q \cdot \tilde{f} \text{ where } Q_{tk} = \frac{1}{\sqrt{n}}\, \omega_n^{kt}\,.$$

Thus we can calculate $\tilde{f}$ from f by multiplying by the inverse of Q,

$$\tilde{f} = Q^{-1} \cdot f\,.$$

As the following exercise shows, Q^{-1} is simply Q's complex conjugate Q^*, i.e., the matrix where we take the complex conjugate of each entry:

Exercise 3.2 *Prove that $Q \cdot Q^* = \mathbb{1}$ where $\mathbb{1}$ denotes the identity matrix.*

Since Q is symmetric, we can also say that Q^{-1} is the transpose of its complex conjugate. Such matrices are called *unitary*, and we will see in Chapter 15 that they play a crucial role in quantum computation.

We can now write

$$\tilde{f} = Q^* \cdot f$$

or, as an explicit sum,

$$\tilde{f}(k) = \frac{1}{\sqrt{n}} \sum_{t=0}^{n-1} f(t)\, \omega_n^{-kt}\,. \tag{3.6}$$

Turning now to algorithms, what is the most efficient way to evaluate the sum (3.6), or its twin (3.5)? We can multiply an n-dimensional vector by an $n \times n$ matrix with n^2 multiplications. Assuming that we do our arithmetic to some constant precision, each multiplication takes constant time. Thus we can obtain $\tilde{f}$ from f, or vice versa, in $O(n^2)$ time. Can we do better?

Indeed we can. After all, Q is not just any $n \times n$ matrix. It is highly structured, and we can break the product $Q^* \cdot f$ down in a recursive way. We again divide and conquer, by dividing the list of samples $f(t)$ into two sublists of size $n/2$: those where t is even, and those where it is odd. Assume n is even, and define

$$f_{\text{even}}(s) = f(2s) \text{ and } f_{\text{odd}}(s) = f(2s+1),$$

where s ranges from 0 to $n/2-1$. Also, write

$$k = (n/2)k_0 + k',$$

where $k_0 = 0$ or 1 and k' ranges from 0 to $n/2-1$. In particular, if n is a power of 2 and we write k in binary, then b is the most significant bit of k, and k' is k with this bit removed.

Now we can separate the sum (3.6) into two parts as follows:

$$\begin{aligned}
\tilde{f}(k) &= \frac{1}{\sqrt{n}} \left(\sum_{t \text{ even}} f(t)\, \omega_n^{-kt} + \sum_{t \text{ odd}} f(t)\, \omega_n^{-kt} \right) \\
&= \frac{1}{\sqrt{n}} \left(\sum_{s=0}^{n/2-1} f_{\text{even}}(s)\, \omega_n^{-2ks} + \omega_n^{-k} \sum_{s=0}^{n/2-1} f_{\text{odd}}(s)\, \omega_n^{-2ks} \right) \\
&= \frac{1}{\sqrt{2}} \frac{1}{\sqrt{n/2}} \left(\sum_{s=0}^{n/2-1} f_{\text{even}}(s)\, \omega_{n/2}^{-k's} + (-1)^{k_0}\, \omega_n^{-k'} \sum_{s=0}^{n/2-1} f_{\text{odd}}(s)\, \omega_{n/2}^{-k's} \right) \\
&= \frac{1}{\sqrt{2}} \left(\tilde{f}_{\text{even}}(k') + (-1)^{k_0}\, \omega_n^{-k'}\, \tilde{f}_{\text{odd}}(k') \right).
\end{aligned} \tag{3.7}$$

We used the following facts in the third line,

$$\begin{aligned}
\omega_n^2 &= \omega_{n/2} \\
\omega_{n/2}^{-k} &= \mathrm{e}^{-2\imath\pi k_0}\, \omega_{n/2}^{-k'} = \omega_{n/2}^{-k'} \\
\omega_n^{-k} &= \mathrm{e}^{-\imath\pi k_0}\, \omega_{n/2}^{-k'} = (-1)^{k_0}\, \omega_{n/2}^{-k'}.
\end{aligned}$$

Equation (3.7) gives us our divide-and-conquer algorithm. First, we recursively calculate the Fourier transforms $\tilde{f}_{\text{even}}$ and $\tilde{f}_{\text{odd}}$. For each of the n values of k, we multiply $\tilde{f}_{\text{odd}}(k')$ by the "twiddle factor" $\omega_n^{-k'}$. Finally, depending on whether k_0 is 0 or 1, we add or subtract the result from $\tilde{f}_{\text{even}}(k')$ to obtain $\tilde{f}(k)$.

Let $T(n)$ denote the time it takes to do all this. Assuming again that we do our arithmetic to fixed precision, $T(n)$ is the time $2T(n/2)$ it takes to calculate $\tilde{f}_{\text{even}}$ and $\tilde{f}_{\text{odd}}$, plus $O(1)$ for each application of (3.7). This gives

$$T(n) = 2T(n/2) + \Theta(n).$$

If we assume that n is a power of 2 so that it is even throughout the recursion, then $T(n)$ has the same kind of scaling as `Mergesort`,

$$T(n) = \Theta(n \log n).$$

This is known as the Fast Fourier Transform, or FFT for short.

What if n is not a power of 2? If n is composite, the divide-and-conquer idea still works, since if n has a factor p, we can divide the list into p sublists of size n/p. As Problem 3.14 shows, this gives a running time of $\Theta(n \log n)$ whenever n is a so-called "smooth" number, one whose largest prime factor is bounded by some constant. Another type of FFT, described in Problem 3.17, works when n is prime. Thus we can achieve a running time of $\Theta(n \log n)$ for any n.

The FFT plays an important role in astronomy today, but in ways that Ptolemy could never have imagined—searching for extrasolar planets, finding irregularities in the cosmic microwave background, and listening for gravitational waves. And as we will see in Chapter 15, a quantum version of the FFT is at the heart of Shor's quantum algorithm for FACTORING.

3.5

3.3 Dynamic Programming

> Turning to the succor of modern computing machines, let us renounce all analytic tools.
>
> Richard Bellman, *Dynamic Programming*

Most problems are hard because their parts interact—each choice we make about one part of the problem has wide-ranging and unpredictable consequences in the other parts. Anyone who has tried to pack their luggage in the trunk of their car knows just what we mean.

This makes it hard to apply a divide-and-conquer approach, since there is no obvious way to break the problem into subproblems that can be solved independently. If we try to solve the problem a little at a time, making a sequence of choices, then each such sequence creates a different subproblem that we have to solve, forcing us to explore an exponentially branching tree of possible choices.

However, for some problems these interactions are limited in an interesting way. Rather than each part of the problem affecting every other part in a global fashion, there is a relatively narrow channel through which these interactions flow. For instance, if we solve part of the problem, the remaining subproblem might be a function, not of the entire sequence of choices we have made up to this point, but only of the most recent one. As a consequence, many sequences of choices lead to the same subproblem. Once we solve this subproblem, we can reuse its solution elsewhere in the search tree. This lets us "fold up" the search tree, collapsing it from exponential to polynomial size.

This may all seem a bit abstract at this point, so let's look at two examples: typesetting books and aligning genomes.

3.3.1 *Moveable Type*

Anyone who would letterspace lower case would steal sheep.
Frederic Goudy

The book you hold in your hands owes much of its beauty to the TEX typesetting system. One of the main tasks of such a system is to decide how to break a paragraph into lines. Once it chooses where the line breaks go, it "justifies" each line, stretching the spaces between words so that the left and right margins are straight. Another option is to stretch out the spaces between the letters of each word, a practice called *letterspacing*, but we agree with the great font designer Mr. Goudy that this is an abomination.

Our goal is to place the line breaks in a way that is as aesthetic as possible, causing a minimum of stretching. For simplicity, we will ignore hyphenation, so that each line break comes between two words. Thus given a sequence of words $w_1,\ldots,w_n$, we want to choose locations for the line breaks. If there are $\ell+1$ lines, we denote the line breaks $j_1,\ldots,j_\ell$, meaning that there is a line break after w_{j_1}, after w_{j_2}, and so on. Thus the first line consists of the words $w_1,\ldots,w_{j_1}$, the second line consists of the words $w_{j_1+1},\ldots,w_{j_2}$, and so on.

Justifying each of these lines has an aesthetic cost. Let $c(i,j)$ denote the cost of putting the words $w_i,\ldots,w_j$ together on a single line. Then for a given set of line breaks, the total cost is

$$c(1,j_1)+c(j_1+1,j_2)+\cdots+c(j_\ell+1,n). \tag{3.8}$$

How might $c(i,j)$ be defined? If it is impossible to fit $w_i,\ldots,w_j$ on a single line, because their total width plus the $j-i$ spaces between them exceeds the width of a line, we define $c(i,j)=\infty$. If, on the other hand, we can fit them one a line with some room to spare, we define $c(i,j)$ as some increasing function of the amount E of extra room. Let L be the width of a line, and suppose that a space has width 1. Then

$$\begin{aligned} E &= L-\big(|w_i|+1+|w_{i+1}|+1+\cdots+1+|w_j|\big) \\ &= L-(j-i)-\sum_{t=i}^{j}|w_t|, \end{aligned}$$

where $|w_i|$ denotes the width of w_i. A typical cost function, which is not too different from that used in TEX, is

$$c(i,j)=\left(\frac{E}{j-i}\right)^3.$$

Here $E/(j-i)$ is the factor by which we have to stretch each of the $j-i$ spaces to fill the line, and the exponent 3 is just an arbitrary way of preferring smaller stretch factors. This formula is undefined if $i=j$, i.e., if the line contains just a single word, and it also ignores the fact that we don't justify the last line of the paragraph. However, it is good enough to illustrate our algorithm.

With this definition of $c(i,j)$, or any other reasonable definition, our goal is to choose the line breaks so that the total cost (3.8) is minimized:

TYPESETTING
Input: A sequence of words $w_1,\ldots,w_n$
Output: A sequence of line break locations $j_1,\ldots,j_\ell$ that minimizes the total cost of the paragraph

How can we solve this problem in polynomial time?

We could try a *greedy* algorithm, in which we fill each line with as many words as possible before moving on to the next. However, this is not always the right thing to do. Consider the following paragraph, which we have typeset greedily:

Daddy, please buy me a
little baby
humuhumunukunukuāpuaʻa!

Arguably, this looks better if we stretch the first line more in order to reduce the space on the second line,

Daddy, please buy
me a little baby
humuhumunukunukuāpuaʻa!

This is precisely the kind of sacrifice that a greedy algorithm would refuse to make. The difficulty is that the different parts of the problem interact—the presence of a word on the last line can change how we should typeset the first line.

So, what kind of algorithm might work? The number of possible solutions is 2^n, since we could in theory put a line break, or not, after every word. Of course, this includes absurd solutions such as putting every word on its own line, or putting the entire paragraph on a single line. But even if we already know that there are ℓ line breaks and $\ell+1$ lines, finding the optimal locations $j_1,\ldots,j_\ell$ is an ℓ-dimensional optimization problem. If $\ell = n/10$, say, corresponding to an average of 10 words per line, there are $\binom{n}{n/10} = 2^{\Omega(n)}$ possible solutions. Thus, *a priori*, this problem seems exponentially hard.

The key insight is that each line break cuts the paragraph into two parts, which can then be optimized independently. In other words, each line break blocks the interaction between the words before it and those after it. This doesn't tell us where to put a line break, but it tells us that once we make this choice, it cuts the problem into separate subproblems.

So, where should the first line break go? The cost of the entire paragraph is the cost of the first line, plus the cost of everything that comes after it. We want to minimize this total cost by putting the first line break in the right place. Let $f(i)$ denote the minimum cost of typesetting the words $w_i,\ldots,w_n$, so that $f(1)$ is the minimum cost of the entire paragraph. Then

$$f(1) = \min_j \big(c(1,j) + f(j+1)\big)\,.$$

The same argument applies to the optimal position of the *next* line break, and so on. In general, if we have decided to put a line break just before the ith word, the minimum cost of the rest of the paragraph is

$$f(i) = \min_{j:i\le j<n} \big(c(i,j) + f(j+1)\big)\,, \tag{3.9}$$

and the optimal position of the next line break is whichever j minimizes this expression. The base case is $f(n+1)=0$, since at that point there's nothing left to typeset. This gives us a recursive algorithm for $f(i)$, which we show in Figure 3.9.

Exercise 3.3 *Modify this algorithm so that it returns the best location j of the next line break.*

```
f(i) // the minimum cost of the paragraph starting with the ith word
begin
    if i = n+1 then return 0 ;
    f_min := +∞ ;
    for j = i to n do
        f_min := min(f_min, c(i,j) + f(j+1)) ;
    return f_min ;
end
```

FIGURE 3.9: The recursive algorithm for computing the minimum cost $f(i)$ of typesetting the part of the paragraph starting with the ith word.

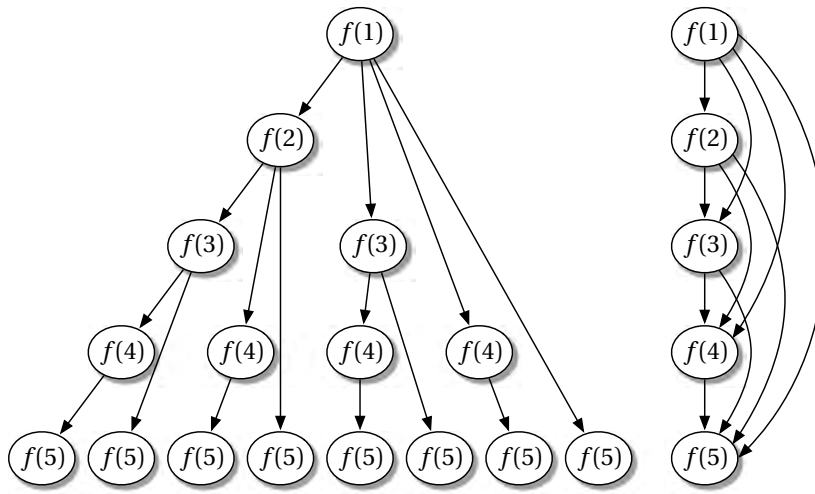


FIGURE 3.10: Calculating the optimal cost $f(1)$ using the recursive algorithm for TYPESETTING causes us to solve the same subproblems many times, taking exponential time. But if we memorize our previous results, we solve each subproblem only once. Here $n = 5$.

This is all well and good. However, if we're not careful, this algorithm will take exponential time. The reason is that it recalculates the same values of f many times. For instance, it calculates $f(4)$ whenever it calculates $f(1)$, $f(2)$, or $f(3)$; it calculates $f(3)$ whenever it calculates $f(1)$ or $f(2)$; and so on.

Exercise 3.4 *Show that the recursive algorithm for $f(1)$ will calculate $f(n)$ a total of 2^{n-2} times.*

To avoid this, we *memorize* values of the function we have already calculated, by placing each one in a table. Then, when we call the function for $f(i)$, we first check the table to see if we have already calculated it, and we only launch our recursive scheme if we haven't. By storing our previous results, we only have to calculate each $f(i)$ once (see Figure 3.10). Since there are n different values of i and the **for** loop runs through $O(n)$ values of j for each one, the total running time is $O(n^2)$.

This combination of recursion and memorization implements the algorithm from the top down. Another approach is to work from the bottom up—solving the simplest subproblems first, then the sub-

problems that depend on these, and so on. In this case, we would work backwards, first calculating $f(n)$, then $f(n-1)$, and so on until we reach the cost $f(1)$ of the entire paragraph. Programs based on these two implementations look rather different from each other, but the actual work done by the computer is essentially the same.

To summarize, the parts of the paragraph in TYPESETTING interact, but each line break blocks this interaction, and cuts the problem into independent subproblems. If we have chosen how to typeset part of the paragraph, then the subproblem we have left, namely typesetting the rest of the paragraph $w_i, \ldots, w_n$, depends only on our last choice of line break $j = i-1$, and not on the entire sequence of choices before that. The total number of different subproblems we have to deal with is poly(n), one for each possible value of i, and it takes poly(n) time to combine these subproblems at each stage of the recursion. Thus if we save our previous results, solving each subproblem only once and reusing it when it appears again in the tree, the total running time is poly(n), and TYPESETTING is in P.


3.6

3.3.2 *Genome Alignment*

Let's use dynamic programming to solve another problem, which is important in genomics, spell checking, and catching term-paper plagiarism. Given two strings s and t, the *edit distance* $d(s,t)$ between them is the minimum number of insertions, deletions, or mutations needed to change s to t, where each mutation changes a single symbol. For instance, consider the following two important professions:

```
P A S T R Y C O   O K
  A S T R     O N O M E R
```

Such an arrangement of the two strings, showing which symbols correspond to each other and which are inserted or deleted, is called an *alignment*. Since this alignment involves deleting `P`, `Y`, and `C`, inserting `N`, `E`, and `R`, and mutating `K` to `M`, it shows that the edit distance between these two strings is at most 7. But how can we tell whether this is optimal?

Let's call EDIT DISTANCE and ALIGNMENT the problems of finding the edit distance and the optimal alignment respectively. Just as in TYPESETTING, the number of possible alignments grows exponentially in the length of the strings:

Exercise 3.5 *Suppose that s and t each have length n. Show that the number of possible alignments between them is at least 2^n. If you enjoy combinatorics, show that it is*

$$\sum_{j=0}^{n} \binom{n}{j}^2 = \binom{2n}{n}.$$

Hint: each alignment specifies a subset of the symbols of s, and a corresponding subset of the symbols of t.

So, *a priori*, it is not obvious that we can solve these problems in polynomial time.

However, we can again use dynamic programming for the following reason. Suppose we cut s into two parts, s_{left} and s_{right}. In the optimal alignment, this corresponds to cutting t somewhere, into t_{left} and t_{right}. Once we decide where the corresponding cut is, we can then find the optimal alignment of s_{left} with t_{left}, and, independently, the optimal alignment of s_{right} with t_{right}. Just as placing a line break separates

a paragraph into two independent parts, making an initial choice about the alignment cuts it into two independent subproblems.

In particular, if we decide how to align the very beginning of the two strings, we can find the alignment between their remainders separately. So let s_1 denote the first symbol of s, let s' be the remainder of s with s_1 removed, and define t_1 and t' similarly. Now there are three possibilities. Either the optimal alignment consists of deleting s_1 and aligning s' with t; or it inserts t_1 and aligns s with t'; or it matches s_1 and t_1, mutating one into the other if they are different, and aligns s' with t'. The edit distance in the first two cases is $d(s',t)+1$ or $d(s,t')+1$ respectively. In the last case, it is $d(s',t')$ if s_1 and t_1 are the same, and $d(s',t')+1$ if they are different.

Since the edit distance is the minimum of these three possibilities, this gives us a recursive equation,

$$d(s,t) = \min\left(d(s',t)+1, d(s,t')+1, \begin{cases} d(s',t') & \text{if } s_1 = t_1 \\ d(s',t')+1 & \text{if } s_1 \neq t_1 \end{cases}\right). \tag{3.10}$$

Evaluating (3.10) gives us a recursive algorithm, which calculates the edit distance $d(s,t)$ in terms of the edit distance of various substrings. Just as for our TYPESETTING algorithm, we need to memorize the results of subproblems we have already solved. As the following exercise shows, the number of different subproblems is again polynomial, so dynamic programming yields polynomial-time algorithms for EDIT DISTANCE and ALIGNMENT.

Exercise 3.6 *Show that if s and t are each of length n, there are only $O(n^2)$ different subproblems that we could encounter when recursively calculating $d(s,t)$.*

Exercise 3.7 *Write a recursive algorithm with memorization that outputs the optimal alignment of s and t, using an algorithm for the edit distance $d(s,t)$ as a subroutine.*

Both TYPESETTING and ALIGNMENT have a one-dimensional character, in which we solve a problem from left to right. The subproblem we have left to solve only "feels" our rightmost, or most recent, choices. Thus, while there are an exponential number of ways we could typeset the first half of a paragraph, or align the first halves of two strings, many of these lead to exactly the same remaining subproblem, and the number of different subproblems we ever need to consider is only polynomial.

To state this a little more abstractly, let's visualize the interactions between the parts of a problem as a graph. If this graph consists of a one-dimensional string, cutting it anywhere separates it into two pieces. Moreover, there are n places to cut a string of length n, and there are a polynomial number of substrings that can result from these cuts. For this reason, many problems involving strings or sequences can be solved by dynamic programming.

Strings are not the only graphs that can be cut efficiently in this way. As Problems 3.25, 3.26, and 3.28 show, dynamic programming can also be used to solve problems on trees and even on certain fractals. However, if the network of interactions between parts of a problem is too rich, it is too hard to cut into subproblems, and dynamic programming fails to give a polynomial-time algorithm.

3.7

3.4 Getting There From Here

> Go often to the house of thy friend, for weeds soon choke up the unused path.
>
> Scandinavian Proverb

Imagine that we are living on a graph. We start at one vertex, and we want to reach another. We ask a friendly farmer the classic question: can we get there from here?

REACHABILITY
Input: A (possibly directed) graph G and two vertices s, t
Question: Is there a path from s to t?

Now suppose the graph is *weighted*, so that for each pair of vertices i and j, the edge between them has a length w_{ij}. Then what is the shortest path from s to t?

SHORTEST PATH
Input: A weighted graph G and two vertices s, t
Question: How long is the shortest path from s to t?

Note that "shortest" here could also mean cheapest or fastest, if w_{ij} is measured in dollars or hours rather than in miles.

There are many ways to solve these two problems, including some of the algorithmic strategies we have already seen. However, so many other problems can be expressed in terms of REACHABILITY and SHORTEST PATH that they deserve to be considered algorithmic strategies in and of themselves. In this section we look at them from several points of view.

3.4.1 Exploration

Perhaps the simplest way to solve REACHABILITY is to start at the source s and explore outward, marking every vertex we can reach, until we have exhausted every possible path. This naive approach gives the algorithm `Explore` shown in Figure 3.11. At each step, we have a set Q of vertices waiting to be explored. We take a vertex u from Q, mark it, and add its unmarked neighbors to Q. When Q is empty, our exploration is complete, and we can check to see if t is marked.

The order in which `Explore` explores the graph depends on which vertex u we remove from Q, and this depends on what kind of data structure Q is. If Q is a *stack*, like a stack of plates, then it acts in a last-in, first-out way. When we ask it for a vertex u, it returns the one at the top, which is the one that was added most recently. In this case, `Explore` performs a *depth-first* search, pursuing each path as deeply as possible, following it until it runs out of unmarked vertices. It then backtracks to the last place where it had a choice, pursues the next path as far as possible, and so on.

Depth-first searches are easy to express recursively. As the program in Figure 3.12 calls itself, its children explore u's neighbors, its grandchildren explore u's neighbors' neighbors, and so on.

```
Explore(G, s, t)
input: a graph G and a vertex s
begin
    Q := {s} ;
    while Q is nonempty do
        remove a vertex u from Q ;
        mark u ;
        for all unmarked neighbors v of u do add v to Q
    end
end
```

FIGURE 3.11: This algorithm explores the graph and marks every vertex we can reach from s.

```
Explore(G, u)
begin
    mark u ;
    for all unmarked neighbors v of u do Explore(G, v)
end
```

FIGURE 3.12: A depth-first search exploration of the graph, written recursively.

On the other hand, Q could be a *queue*, like a line of people waiting to enter a theater. A queue operates in a first-in, first-out fashion, so the next u in line is the vertex that has been waiting in Q the longest. In that case, `Explore` performs a *breadth-first* search. It explores all of s's neighbors first, then s's neighbors' neighbors, and so on, expanding the set of marked vertices outward one layer at a time. If G is an unweighted graph, in which every edge has weight 1, the paths that `Explore` follows are among the shortest paths from s as shown in Figure 3.13.

3.4.2 Middle-First Search

Let's look at a rather different kind of algorithm for REACHABILITY. Recall from Section 2.4.2 that the *adjacency matrix* of a graph with n vertices is an $n \times n$ matrix A, where $A_{ij} = 1$ if there is an edge from i to j and 0 otherwise. Now consider the following useful fact, where $A^t = A \cdot A \cdots A$ (t times) denotes the tth matrix power of A:

$$(A^t)_{ij} \text{ is the number of paths of length } t \text{ from } i \text{ to } j\,.$$

For example,

$$(A^3)_{ij} = \sum_{k,\ell} A_{ik} A_{k\ell} A_{\ell j}$$

is the number of pairs k, ℓ such that there are edges from i to k, from k to ℓ, and from ℓ to j. Equivalently, it is the number of paths of the form $i \to k \to \ell \to j$. Consider the graph in Figure 3.14. The powers of its adjacency matrix are

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad A^2 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}, \quad A^3 = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}.$$

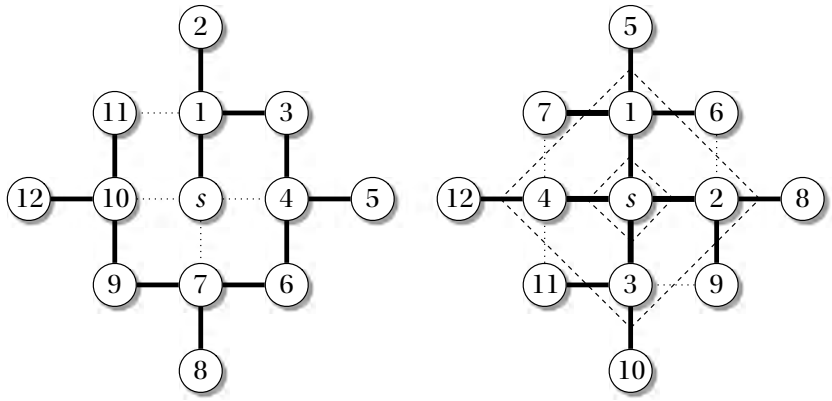


FIGURE 3.13: Left, a depth-first search which follows a path as deeply as possible, and then backtracks to its last choice. Right, a breadth-first search which builds outward from the source one layer at a time. The neighbors of each vertex are ordered north, east, south, west, and vertices are numbered in the order in which they are removed from the stack or queue.

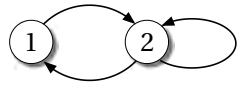


FIGURE 3.14: A little directed graph.

For instance, there are 3 paths of length 3 that begin and end at vertex 2.

If the graph does not already have *self-loops*—that is, edges from each vertex to itself, giving us the option of staying put on a given step—we can create them by adding the identity matrix $\mathbb{1}$ to A. Now consider the following exercise:

Exercise 3.8 *Given a graph G with n vertices and adjacency matrix A, show that there is a path from s to t if and only if $(\mathbb{1}+A)^{n-1}_{st}$ is nonzero.*

This offers us a nice way to solve REACHABILITY on all pairs of vertices s, t at once: simply raise the matrix $\mathbb{1}+A$ to the $n-1$st power. For simplicity, we might as well raise it to the nth power instead.

What is the most efficient way to compute $(\mathbb{1}+A)^n$? We could start with $B = \mathbb{1}$ and then repeat the following n times,

$$B \to B(\mathbb{1}+A).$$

This is a bit like breadth-first search, since each update extends the set of reachable vertices by another step. However, this will involve n matrix multiplications. A smarter approach is to start with $B = \mathbb{1}+A$ and square the matrix repeatedly:

$$B \to B^2,$$

or more explicitly,

$$B_{ij} \to \sum_k B_{ik} B_{kj}. \tag{3.11}$$

Just as in our divide-and-conquer algorithm for exponentiation in Section 3.2.2, we only need to repeat this update $\log_2 n$ times to reach the nth power, rounding up if n is not a power of 2.

What if we just want a Boolean matrix, where $B_{ij} = 1$ or 0 depending on whether there is a path from i to j or not, rather than counting paths of various lengths? We can do this in the style of (3.11), by replacing multiplication with AND and addition with OR. This gives

$$B_{ij} \to \bigvee_k (B_{ik} \wedge B_{kj}). \tag{3.12}$$

Here $\bigvee_k$ means the OR over all k, just as $\sum_k$ means the sum. This equation says, recursively, that we can get from i to j if there is a k such that we can get from i to k and from k to j. The base case of this recursion is the initial value $B = \mathbb{1} \vee A$—that is, j is reachable from i if $i = j$ or i and j are adjacent.

Since this strategy tries to find a vertex in between i and j, it is often called *middle-first search*. While it is not the most efficient in terms of time, the fact that it only needs $\log_2 n$ levels of recursion makes it very efficient in its use of memory. In Chapter 8, we will apply middle-first search to graphs of exponential size, whose vertices correspond to every possible state of a computer.

3.4.3 Weighty Edges

Now let's put a cost or length w_{ij} on each edge, and ask for a matrix B whose entries B_{ij} are the lengths of the shortest paths. This variant of the problem deserves a name:

ALL-PAIRS SHORTEST PATHS
Input: A weighted graph G with weights w_{ij} on each edge (i, j)
Output: A matrix where B_{ij} is the length of the shortest path from i to j

We start by defining w_{ij} for all pairs i, j, instead of just those connected by an edge of G. If $i \neq j$ and there is no edge from i to j, we set $w_{ij} = \infty$. Similarly, we set $w_{ii} = 0$ since it costs nothing to stay put. This gives us an $n \times n$ matrix W.

Now, in analogy to our matrix-product algorithm for REACHABILITY, we initially set $B_{ij} = w_{ij}$ for all i, j, and then "square" B repeatedly. But this time, we replace multiplication with addition, and replace addition with minimization:

$$B_{ij} \to \min_k (B_{ik} + B_{kj}). \tag{3.13}$$

In other words, the length B_{ij} of the shortest path from i to j is the minimum, over all k, of the sum of the lengths of the shortest paths from i to k and from k to j. We claim that, as before, we just need to update B according to (3.13) $\log_2 n$ times in order to get the correct value of B_{ij} for every pair.

This gives the algorithm shown in Figure 3.15 for ALL-PAIRS SHORTEST PATHS. For clarity, $B(m)$ denotes the value of B after m iterations of (3.13), so $B(0) = W$ and our final result is $B(\log_2 n)$. Assuming that min and + take $O(1)$ time, it is easy to see from these nested loops that this algorithm's total running time is $\Theta(n^3 \log n)$. Problem 3.34 shows how to improve this to $\Theta(n^3)$ by arranging these loops a little differently.

3.8

In a sense, this algorithm is an example of dynamic programming. Once we have chosen a midpoint k, finding the shortest path from i to j breaks into two independent subproblems—finding the shortest path from i to k and the shortest path from k to j. The pseudocode we give here is a "bottom-up" implementation, in which we calculate $B(m)$ from the previously-calculated values $B(m-1)$.

```
All-Pairs Shortest Paths(W)
input: a matrix W of weights w_ij
output: a matrix B where B_ij is the length of the shortest path from i to j
begin
    forall the i, j do B_ij(0) := w_ij ;
    for m = 1 to log_2 n do
        for i = 1 to n do
            for j = 1 to n do
                B_ij(m) := B_ij(m-1) ;
                for k = 1 to n do
                    B_ij(m) := min(B_ij(m), B_ik(m-1) + B_kj(m-1));
    return B(log_2 n) ;
end
```

FIGURE 3.15: A middle-first algorithm for ALL-PAIRS SHORTEST PATHS. To distinguish each value of B from the previous one, we write $B(m)$ for the mth iteration of (3.13).

There is another reason that this algorithm deserves the name "dynamic." Namely, we can think of (3.13) as a *dynamical system*, which starts with the initial condition $B = W$ and then iterates until it reaches a fixed point.

Exercise 3.9 *Show how to modify these algorithms for* REACHABILITY *and* SHORTEST PATH *so that they provide the path in question, rather than just its existence or its length. Hint: consider an array that records, for each i and j, the vertex k that determined the value of B_{ij} in* (3.12) *or* (3.13).

Exercise 3.10 *Suppose some of the weights w_{ij} are negative. Can we still find the lengths of the shortest paths—some of which may be negative—by iterating* (3.13) *until we reach a fixed point? What happens if there is a* cycle *whose total length is negative?*

3.4.4 But How do we Know it Works?

Up to now, the *correctness* of our algorithms—that is, the fact that they do what they are supposed to do—has been fairly self-evident. As our algorithms get more complicated, it's important to discuss how to prove that they actually work.

Typically, these proofs work by induction on the number of layers of recursion, or the number of times a loop has run. We would like to establish that, after the algorithm has reached a certain stage, it has made some concrete type of progress—that it has solved some part of the problem, or reached a solution of a certain quality. Such a partial guarantee is often called a *loop invariant*. The next exercise asks you to use this approach to prove that our algorithm for ALL-PAIRS SHORTEST PATHS works.

Exercise 3.11 *Prove that, during our algorithm for* ALL-PAIRS SHORTEST PATHS, *$B_{ij}(m)$ is always an upper bound on the length of the shortest path from i to j. Then, show by induction on m that this algorithm satisfies the following loop invariant: after running the outermost loop m times, $B_{ij}(m)$ equals the length of the shortest path from i to j which takes 2^m or fewer steps in the graph. Conclude that as soon as $2^m \geq n$, the algorithm is complete.*

We conclude this section by urging the reader to solve Problems 3.20 and 3.21. These problems show that TYPESETTING and EDIT DISTANCE can both be recast in terms of SHORTEST PATH, demonstrating that SHORTEST PATH is capable of expressing a wide variety of problems.

3.5 When Greed is Good

We turn now to our next algorithmic strategy: greed. Greedy algorithms solve problems step-by-step by doing what seems best in the short term, and never backtracking or undoing their previous decisions. For many problems, this is a terrible idea, as current economic and environmental policies amply demonstrate, but sometimes it actually works. In this section, we look at a greedy algorithm for a classic problem in network design, and place it in a general family of problems for which greedy algorithms succeed.

3.5.1 Minimum Spanning Trees

> The trees that are slow to grow bear the best fruit.
>
> Molière

In the 1920s, Jindřich Saxel contacted his friend, the Czech mathematician Otakar Borůvka, and asked him how to design an efficient electrical network for South Moravia (an excellent wine-growing region). This led Borůvka to the following problem: we have a graph where each vertex is a city, and each edge e has a length or cost $w(e)$. We want to find a subgraph T that *spans* the graph, i.e., that connects all the vertices together, with the smallest total length $w(T) = \sum_{e \in T} w(e)$.

If the edge between two cities is part of a cycle, we can remove that edge and still get from one city to the other by going around the other way—so the minimum spanning subgraph has no cycles. Since a graph without cycles is a tree, what we are looking for is a *minimum spanning tree*. Thus Borůvka's problem is

MINIMUM SPANNING TREE
Input: A weighted connected graph $G = (V, E)$
Question: A spanning tree T with minimum total weight $w(T)$

We assume that G is connected, since otherwise asking for a spanning tree is a bit unfair.

How can we find the minimum spanning tree? Or, if there is more than one, one of the minimum ones? Once again, the number of possible solutions is exponentially large. If G is the *complete* graph on n vertices, in which all $\binom{n}{2}$ pairs of vertices have edges between them—as in Borůvka's original problem, since we could choose to lay an electrical cable between any pair of cities—then Problem 3.36 shows that the number of possible spanning trees is n^{n-2}. For $n = 100$ this is larger than the number of atoms in the known universe, so we had better find some strategy other than exhaustive search.

Let's try a greedy approach. We grow the network step-by-step, adding edges one at a time until all the vertices are connected. We never add an edge between two vertices that are already connected to each other—equivalently, we never complete a cycle. Finally, we start by adding the lightest edges, using the heavier ones later if we have to. This gives *Kruskal's algorithm*, shown in Figure 3.17.

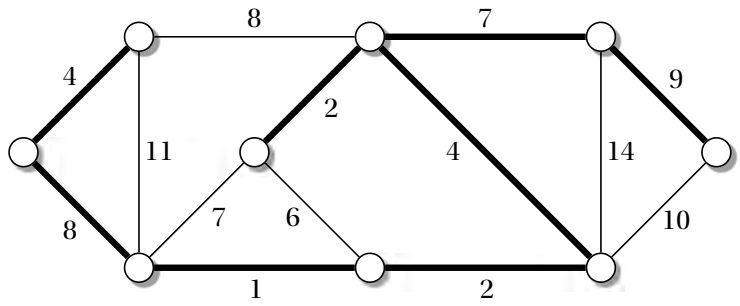


FIGURE 3.16: A weighted graph and a minimum spanning tree. Is it unique?

```
Kruskal(G)
input: a weighted graph G = (V, E)
output: a minimum spanning tree
begin
    F := ∅ ;
    sort E in order from lightest to heaviest ;
    for each edge e ∈ E do
        if adding e to F would not complete a cycle then add e to F ;
    return F;
end
```

FIGURE 3.17: Kruskal's algorithm for MINIMUM SPANNING TREE.

Let's prove that this algorithm works. At each step, the network F is a disjoint union of trees, which we call a *forest*. We start with no edges at all, i.e., with a forest of n trees, each of which is an isolated vertex. As F grows, we merge two trees whenever we add an edge between them, until F consists of one big tree. The following exercise shows that F spans the graph by the time we're done:

Exercise 3.12 *Show that if we complete the* ***for*** *loop and go through the entire list of edges, then F is a spanning tree.*

Alternatively, we can stop as soon as F contains $n-1$ edges:

Exercise 3.13 *Show that a forest with n vertices is a spanning tree if and only if it has $n-1$ edges.*

So, Kruskal's algorithm finds a spanning tree. But how do we know it finds one of the best ones? We would like to prove that adding the next-lightest edge is never a bad idea—that it's never a good idea to "sacrifice" by adding a heavier edge now in order to reduce the total weight later. We can prove this inductively using the following lemma.

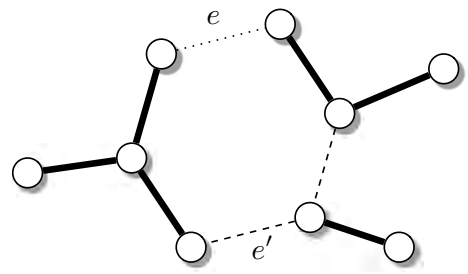


FIGURE 3.18: The proof of Lemma 3.1. Edges already in F are shown in bold, and additional edges in the spanning tree T are dashed. Adding e to T would complete a cycle, and we can obtain a new spanning tree T' by removing e'. If e is one of the lightest edges, T' is at least as light as T.

Lemma 3.1 *Suppose that F is a forest defined on the vertices of G, and that F is contained in a minimum spanning tree of G. Let e be one of the lightest edges outside F that does not complete a cycle, i.e., that connects one of F's trees to another. Then the forest $F \cup \{e\}$ is also contained in a minimum spanning tree.*

Proof Let T be a minimum spanning tree containing F, and assume that T does not contain e. Then T provides some other route from one end of e to the other, so adding e to T would complete a cycle. As shown in Figure 3.18, we could break this cycle by removing some other edge $e' \notin F$. This would give us a new spanning tree T' that also contains F, namely

$$T' = T \cup \{e\} - \{e'\}.$$

But we know that $w(e) \leq w(e')$ since by hypothesis e is one of the lightest edges we could have added to F. So T' is at least as light as T,

$$w(T') = w(T) + w(e) - w(e') \leq w(T).$$

Either we were wrong to assume that T is a minimum spanning tree, or both T and T' are minimum spanning trees. In either case the lemma is proved. □

Using Lemma 3.1 inductively, we see that at all times throughout Kruskal's algorithm, the forest F is a subgraph of some minimum spanning tree. In the terminology of Section 3.4.4, this is a loop invariant. This invariant holds all the way until F is itself is a spanning tree, so it must be a minimum one.

What is the running time of Kruskal's algorithm? Since adding e to F would complete a cycle if and only if there already a path from one of e's endpoints to the other, we can use one of our polynomial-time algorithms for REACHABILITY to check whether or not we should add e. This is far from the most efficient method, but it does show that MINIMUM SPANNING TREE is in P.

3.9

Exercise 3.14 *Run Kruskal's algorithm on the graph of Figure 3.16.*

Exercise 3.15 *Show that if the weights of the edges are distinct from each other, then the* MINIMUM SPANNING TREE *is unique.*

Exercise 3.16 *Find a polynomial-time algorithm that yields the* maximum*-weight spanning tree, and prove that it works.*

3.5.2 *Building a Basis*

Lemma 3.1 tells us that the greedy strategy for MINIMUM SPANNING TREE never steers us wrong—adding the next-lightest edge never takes us off the path leading to an optimal solution. However, this lemma and its proof seem rather specific to this one problem. Can we explain, in more abstract terms, what it is about the structure of MINIMUM SPANNING TREE that makes the greedy strategy work? Can we fit MINIMUM SPANNING TREE into a more general family of problems, all of which can be solved greedily?

One such family is inspired by finding a basis for a vector space. Suppose I have a list S of n-dimensional vectors. Suppose further that $|S| \geq n$ and that S has rank n, so that it spans the entire vector space. We wish to find a subset $F \subseteq S$ consisting of n linearly independent vectors that span the space as well. We can do this with the following greedy algorithm: start with $F = \emptyset$, go through all the vectors in S, and add each one to F as long as the resulting set is still linearly independent. There is never any need to go back and undo our previous decisions, and as soon as we have added n vectors to F, we're done.

This algorithm works because the property that a subset of S is linearly independent—or "independent" for short—obeys the following axioms. The first axiom allows us to start our algorithm, the second gives us a path through the family of independent sets, and the third ensures that we can always add one more vector to the set until it spans the space.

1. The empty set $\emptyset$ is independent.
2. If X is independent and $Y \subseteq X$, then Y is independent.
3. If X and Y are independent and $|X| < |Y|$, there is some element $v \in Y - X$ such that $X \cup \{v\}$ is independent.

Exercise 3.17 *Prove that these three axioms hold if S is a set of vectors and "independent" means linearly independent.*

A structure of this kind, where we have a set S and a family of "independent" subsets which obeys these three axioms, is called a *matroid*. In honor of the vector space example, an independent set to which nothing can be added without ceasing to be independent is called a *basis*.

What does this have to do with MINIMUM SPANNING TREE? Let S be the set E of edges of a graph, and say a subset $F \subseteq E$ is "independent" if it is a forest, i.e., if it has no cycles. Clearly the first two axioms hold, since the empty set is a forest and any subgraph of a forest is a forest. Proving the third axiom is trickier, but not too hard, and we ask you to do this in Problem 3.40. Thus, in any graph G the family of forests forms a matroid. Finally, a "basis"—a forest to which no edges can be added without completing a cycle—is a spanning tree.

Now suppose that each vector in S has some arbitrary weight, and that our goal is to find a basis whose total weight is as small as possible. We can generalize Kruskal's algorithm as follows: sort the elements of S from lightest to heaviest, start with $F = \emptyset$, and add each $v \in S$ to F if the resulting set is still independent. The following lemma, which generalizes Lemma 3.1 and which we ask you to prove in Problem 3.41, proves that this greedy algorithm works.

Lemma 3.2 *Let S be a set where the family of independent sets forms a matroid. Suppose an independent set F is contained in a minimum-weight basis. Let v be one of the lightest elements of S such that $F \cup \{v\}$ is also independent. Then $F \cup \{v\}$ is also contained in a minimum-weight basis.*

Therefore, for any matroid where we can check in polynomial time whether a given set is independent, the problem of finding the minimum-weight basis is in P.

3.5.3 Viewing the Landscape

A greedy algorithm is like a ball rolling down a hill, trying to find a point of minimum height. It rolls down as steeply as possible, until it comes to rest at a point where any small motion increases its altitude. The question is whether this is the *global* minimum, i.e., the lowest point in the entire world, or merely a *local* minimum where the ball has gotten stuck—whether jumping over a hill, or tunneling through one, could get us to an even lower point.

For some problems, such as MINIMUM SPANNING TREE, the landscape of solutions has one big valley, and a ball will roll straight to the bottom of it. Harder problems have a bumpy landscape, with an exponential number of valleys separated by forbidding mountain crags. In a landscape like this, greedy algorithms yield local optima—solutions such that any small change makes things worse. But in order to find the global optimum, we have to make large, complicated changes. In the absence of a good map of the landscape, or a good insight into its structure, our only recourse is an exhaustive search.

3.10

Of course, for a maximization problem the valleys in this metaphor become mountain peaks. Climbing uphill takes us to a local maximum, but the highest peak may be far off in the mist. Next, we will explore a problem where a kind of greedy algorithm finds the global maximum, but only if we define the landscape in the right way.

3.6 Finding a Better Flow

Wouldn't it be wonderful if you could tell whether your current solution to life's problems is the best possible? And, if it isn't, if you could tell how to improve it? If you could answer these questions efficiently, you could find the optimal solution with a kind of greedy algorithm: start with any solution, and repeatedly improve it until you reach the best possible one. In this section, we will apply this strategy to an important type of network flow problem.

My graduate students, my spouse, and I are looking forward to attending a glorious conference, where we will discuss the latest and most beautiful advances in complexity theory and gorge ourselves on fine food. Money is no object. But unfortunately, I have left the travel arrangements until rather late, and there are only a few seats available on each airplane flight that can get us from our university, through various intermediate cities, to the location of the conference. How many of my students can I get there by one route or another?

We can formalize this problem as follows. I have a network G, namely a directed graph where each edge $e = (u, v)$ has a nonnegative integer capacity $c(e)$. I am trying to arrange a *flow* from a source vertex s to a destination t. This flow consists of assigning an integer $f(e)$ to each edge—the number of students who will take that flight—such that $0 \leq f(e) \leq c(e)$. Just as for electric current, the total flow in and out of any vertex other than s and t must be zero. After all, I don't wish to leave any students or spouses, or pick up new ones, at any of the intervening airports.

My goal is to maximize the total flow out of s and into t, which we call the *value* of the flow. This gives the following problem:

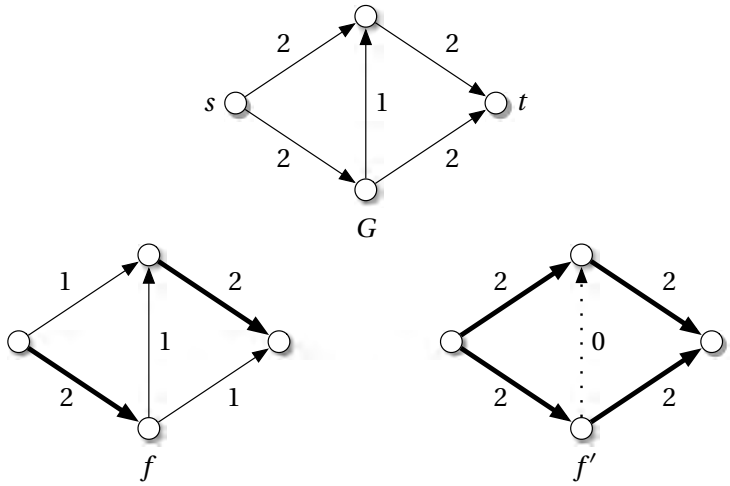


FIGURE 3.19: A network G and two possible flows on it, with edges shown dotted, solid or bold depending on whether the flow on that edge is 0, 1 or 2. On the lower left, a flow f whose value is 3. On the lower right, a better flow f' whose value is 4, which in this case is the maximum.

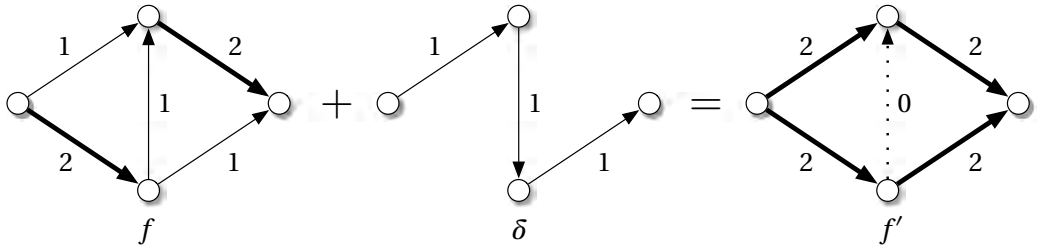


FIGURE 3.20: We can improve the flow, changing f to f', by adding flow along a path δ from s to t. In this case, one of the edges in this path is a reverse edge, and adding flow along it cancels f on the corresponding forward edge.

MAX FLOW
Input: A network G where each edge e has a nonnegative integer capacity $c(e)$, and two vertices s, t
Question: What is the maximum flow from s to t?

As an example, Figure 3.19 shows a simple network, and two flows on it. The flow f shown on the lower left has a value of 3, while the maximum flow f', shown on the lower right, has a value of 4.

Now suppose that our current flow is f. As Figure 3.20 shows, we can improve f by adding flow along a path δ from s to t. When does such a path exist? We can only increase the flow along an edge e if there is unused capacity there. So, given a flow f, let us define a *residual network* G_f where each edge has capacity $c_f(e) = c(e) - f(e)$. If there is a path from s to t along edges with nonzero capacity in G_f, we can increase the flow on those edges.

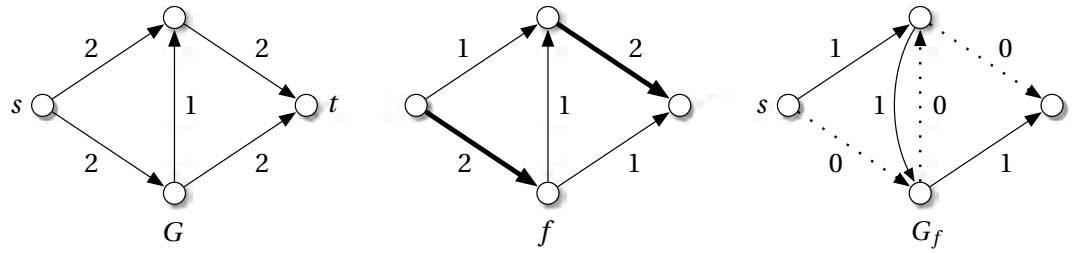


FIGURE 3.21: The residual network G_f for the flow f, showing the unused capacity $c_f(e) = c(e) - f(e)$ of each edge, and a reverse edge $\overline{e}$ with capacity $c_f(\overline{e}) = f(e)$. Other reverse edges are not shown.

However, in Figure 3.20, one of the edges of δ actually goes against the flow in f. By adding flow along this edge, we can cancel some—in this case, all—of the flow on the corresponding edge of G. To allow for this possibility in our residual graph G_f, for each "forward" edge $e = (u, v)$ of G, we also include a "reverse" edge $\overline{e} = (v, u)$. Since we can cancel up to $f(e)$ of the flow along e without making the flow negative, we give these reverse edges a capacity $c_f(\overline{e}) = f(e)$ as shown in Figure 3.21.

Given a flow f, we will call a path from s to t along edges with nonzero capacity in the residual network G_f an *augmenting path.* Now consider the following theorem:

Theorem 3.3 *A flow f is maximal if and only if there is no augmenting path. If there is an augmenting path, increasing the flow along it (and decreasing the flow wherever δ follows reverse edges) produces a new flow f' of greater value.*

Proof First suppose that an augmenting path δ exists. Each edge of δ is either a forward edge e where $f(e) < c(e)$ and $f(e)$ can be increased, or a reverse edge $\overline{e}$ where $f(e) > 0$ and $f(e)$ can be decreased. Thus adding a unit of flow along δ gives a new flow f' with $0 \le f'(e) \le c(e)$ along every edge, and whose value is greater than that of f.

Conversely, suppose that there is a flow f' whose value is greater than that of f. We can define a flow Δ on G_f as follows:

$$\Delta(e) = \max\big(0, f'(e) - f(e)\big)\,,\ \Delta(\overline{e}) = \max\big(0, f(e) - f'(e)\big)\,.$$

In other words, we put flow on e if $f'(e) > f(e)$, and on $\overline{e}$ if $f'(e) < f(e)$.

It is easy to check that the total flow Δ in and out of any vertex other than s or t is zero. Moreover, the flow Δ on each edge of G_f is nonnegative and less than or equal to the capacity c_f, since if $f'(e) > f(e)$ we have

$$0 < \Delta(e) = f'(e) - f(e) \le c(e) - f(e) \le c_f(e),$$

and if $f'(e) < f(e)$ we have

$$0 < \Delta(\overline{e}) = f(e) - f'(e) \le f(e) = c_f(\overline{e}).$$

Thus Δ is a legal flow on G_f. Moreover, Δ has positive value, equal to the value of f' minus that of f. No such flow can exist unless there is a path from s to t along edges with nonzero capacity in G_f, and we are done. □

Theorem 3.3 gives us a simple method for telling whether a flow f is maximal, and to produce an improved flow if it is not. All we have to do is construct the residual network G_f and ask REACHABILITY if there is a path δ from s to t. If there is such a path, we find one, and increase the flow along it as much as possible, i.e., by the minimum of $c_f(e)$ among all the edges $e \in \delta$. We then recalculate the residual capacities in G_f, and continue until no augmenting paths remain. At that point, the current flow is maximal. This is called the *Ford–Fulkerson algorithm*.

The astute reader will immediately ask how many times we need to perform this improvement. Since the value of the flow increases by at least one each time, the number of iterations is at most the sum of all the capacities. This gives us a polynomial-time algorithm if the capacities are only polynomially large. However, in a problem of size n, i.e., described by n bits, the capacities could be n-bit numbers and hence exponentially large as a function of n. As Problem 3.43 shows, in this case the Ford–Fulkerson algorithm could take an exponentially large number of steps to find the maximum flow, if we choose our augmenting paths badly.

Luckily, as Problems 3.44 and 3.45 show, there are several ways to ensure that the total number of iterations is polynomial in n, even if the capacities are exponentially large. One is to use the shortest path from s to t in G_f, and another is to use the "fattest" path, i.e., the one with the largest capacity. Either of these improvements proves that MAX FLOW is in P.

3.11

A priori, one could imagine that the maximum flow is fractional—that at some vertices it splits the flow into noninteger amounts. But the Ford–Fulkerson algorithm gives us the following bonus:

Exercise 3.18 *Prove that if the capacities $c(e)$ are integers, there is a maximal flow f where $f(e)$ is an integer for all e. Hint: use the fact that the Ford–Fulkerson algorithm works.*

Exercise 3.19 *Is the maximal flow always unique? If not, what kind of flow is the difference $\Delta(e)$ between two maximal flows?*

It's interesting to note that if we don't allow reverse edges—if we only allow improvements that increase the flow everywhere along some path in G from s to t—then we can easily get stuck in local optima. In fact, the flow f of Figure 3.19 is a local optimum in this sense. In a sense, reverse edges let us "backtrack" a little bit, and pull flow back out of an edge where we shouldn't have put it.

To put this another way, recall the landscape analogy from Section 3.5.3, but with hills instead of valleys, since this is a maximization problem. Each step of our algorithm climbs from the current flow to a neighboring one, trying to get as high as possible. Without reverse edges, this landscape can be bumpy, with multiple hilltops. By adopting a somewhat larger set of moves, we reorganize the landscape, changing it to a single large mountain that we can climb straight up.

3.7 Flows, Cuts, and Duality

In our last episode, my students and I were trying to attend a conference. Now suppose that an evil competitor of mine wishes to prevent us from presenting our results. He intends to buy up all the empty seats on a variety of flights until there is no way at all to get to the conference, forcing us to seek letters of transit from a jaded nightclub owner. How many seats does he need to buy?

Let's define a *cut* in a weighted graph as a set C of edges which, if removed, make it impossible to get from s to t. The *weight* of the cut is the sum of its edges' weights. Alternately, we can say that a cut is a

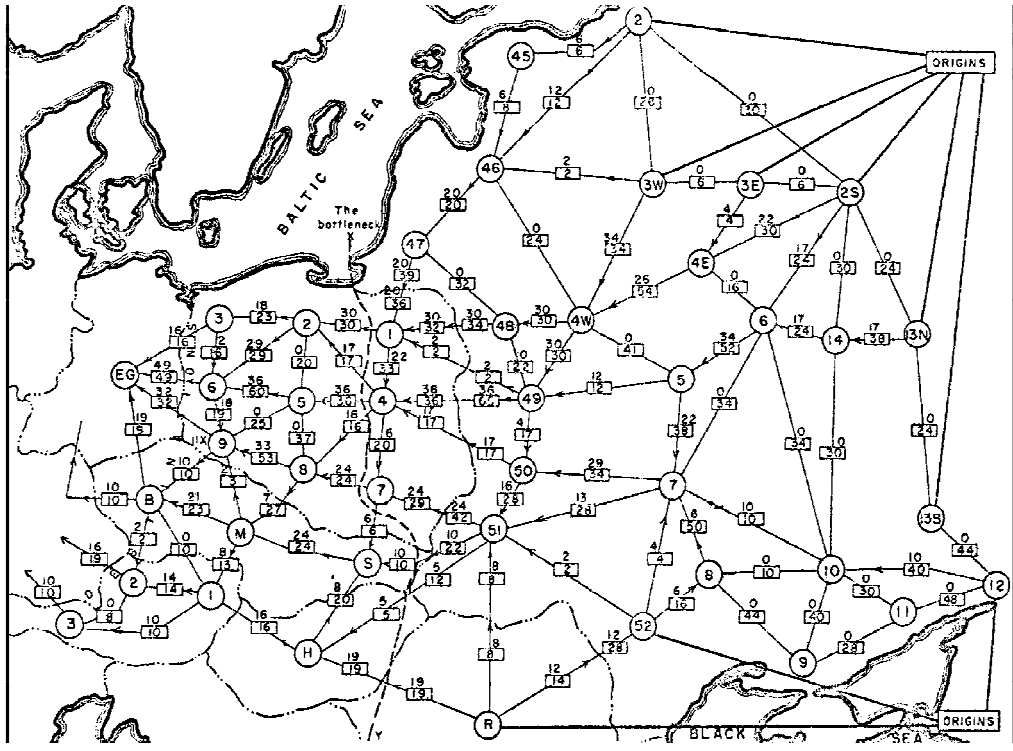


FIGURE 3.22: A MIN CUT problem from the Cold War. A 1955 technical report for the United States Air Force sought to find a "bottleneck" that would cut off rail transport from the Soviet Union to Europe.

partitioning of the vertices of G into two disjoint sets or "sides," S and T, such that $s \in S$ and $t \in T$. Then C consists of the edges that cross from S to T, and its weight is the sum of their capacities.

My competitor wishes to solve the following problem:

MIN CUT (s-t version)
Input: A weighted graph G and two vertices s, t
Question: What is the weight of the minimum cut that separates s from t?

In this section, we will see that MIN CUT and MAX FLOW have exactly the same answer—the weight of the minimum cut is exactly the value of the maximum flow. Thus MIN CUT is really just MAX FLOW in disguise.

To see this, we start by proving that

$$\text{value}(\text{MAX FLOW}) \leq \text{weight}(\text{MIN CUT}). \tag{3.14}$$

Given a cut, the flow has to get from one side to the other, and the flow on any edge is at most its capacity. Therefore, the value of *any* flow is less than or equal to the weight of any cut. In particular, the value of the MAX FLOW is less than or equal to the weight of the MIN CUT.

The tighter statement

$$\text{value}(\text{MAX FLOW}) = \text{weight}(\text{MIN CUT}) \tag{3.15}$$

is less obvious. It certainly holds if G is simply a chain of edges from s to t, since then the MIN CUT consists of an edge with the smallest capacity, and the capacity of this "bottleneck" edge is also the value of the MAX FLOW. Does a similar argument work if there are many paths from s to t, and many vertices where the flow can branch and merge?

Indeed it does, and this follows from the same observations that led to the Ford–Fulkerson algorithm. Recall that Theorem 3.3 shows that if f is a maximal flow, there is no augmenting path. In other words, s is cut off from t by a set of edges whose residual capacity c_f is zero.

Now let S be the set of vertices reachable from s along edges with nonzero c_f, and let T be the rest of the vertices including t. Recall that $c_f(e) = c(e) - f(e)$. Since each edge that crosses from S to T has $c_f(e) = 0$, we have $f(e) = c(e)$. Thus each such edge is *saturated* by the flow f, i.e., used to its full capacity. The weight of the cut between S and T is the total weight of all these edges, and this equals the value of f.

Thus there exists a cut whose weight equals the maximum flow. But since any cut has weight at least this large, this cut is minimal, and (3.15) is proved. Since MAX FLOW is in P, this proves that MIN CUT is in P as well.

Exercise 3.20 *Solve* MIN CUT *in the graph G in Figure 3.19. Is the* MIN CUT *unique? If not, find them all.*

The inequality (3.14) creates a nice interplay between these problems, in which each one acts as a bound on the other. If I show you a cut of weight w, this is a proof, or "witness," that the MAX FLOW has a value at most w. For instance, the set of all edges radiating outward from s form a cut, and indeed no flow can exceed the total capacity of these edges. Conversely, if I show you a flow with value f, this is a proof that the MIN CUT has weight at least f.

From this point of view, (3.15) states that the MIN CUT is the *best possible upper bound* on the MAX FLOW, and the MAX FLOW is the *best possible lower bound* on the MIN CUT. What is surprising is that these bounds are tight, so that they meet in the middle. These two optimization problems have the same solution, even though they are trying to push in opposite directions.

This relationship between MAX FLOW and MIN CUT is an example of a much deeper phenomenon called *duality*. MAX FLOW is a constrained optimization problem, where we are trying to maximize something (the value of the flow) subject to a set of inequalities (the edge capacities). It turns out that a large class of such problems have "mirror images," which are minimization problems analogous to MIN CUT. We will discuss duality in a more general way in Section 9.5.

Finally, it is natural to consider the problem which tries to *maximize*, rather than minimize, the weight of the cut. Now that we know that MIN CUT is in P, the reader may enjoy pondering whether MAX CUT is as well. We will resolve this question, to some extent, in Chapter 5.

3.8 Transformations and Reductions

We close this chapter by introducing a fundamental idea in computational complexity—a transformation, or *reduction*, of one problem to another.

Suppose I am running a dating service. Some pairs of my clients are compatible with each other, and I wish to arrange as many relationships as possible between compatible people. Surprisingly, all my clients are monogamous, and have no interest in ménages à trois. So, my job is to find a set of compatible couples, such that no person participates in more than one couple.

I can represent my clients' compatibilities as a graph $G = (V, E)$, where each client is a vertex and each compatible couple is connected by an edge. I wish to find a *matching*, i.e., a subset $M \subseteq E$ consisting of disjoint edges, which covers as many vertices as possible. If I am especially lucky, there will be a *perfect matching*—an M that covers every vertex in G so that everyone has a partner.

We focus here on the unusual case where all my clients are heterosexual, in which case G is bipartite. This gives me the following problem,

PERFECT BIPARTITE MATCHING
Input: A bipartite graph G
Question: Does G have a perfect matching?

More generally, I want to maximize the number of happy couples by finding the matching M with the largest number of edges:

MAX BIPARTITE MATCHING
Input: A bipartite graph G
Question: What is the maximum matching?

There are an exponential number of possible matchings, and it is not obvious how to find the maximal one in polynomial time. The good news is that we already know how to do this. We just need to translate this problem into another one that we have solved before.

As Figure 3.23 shows, we can transform an instance of MAX BIPARTITE MATCHING into an instance of MAX FLOW. We turn each compatible couple (u, v) into a directed edge $u \to v$, pointing from left to right. We then add two vertices s, t with edges $s \to u$ for each u on the left, and $v \to t$ for each t on the right. Finally, we give every edge in this network a capacity 1.

We claim that the size of the maximum matching equals the value of the MAX FLOW on this network. We express this in the following exercise:

Exercise 3.21 *Show that there is a matching consisting of m edges if and only if this network has a flow of value m. Hint: use the fact proved in Exercise 3.18 that at least one of the maximal flows is integer-valued.*

Thus we can solve MAX BIPARTITE MATCHING by performing this transformation and then solving the resulting instance of MAX FLOW. Since MAX FLOW is in P, and since the transformation itself is easy to do in polynomial time, this proves that MAX BIPARTITE MATCHING and PERFECT BIPARTITE MATCHING are in P.

This type of transformation is called a *reduction* in computer science. On one level, it says that we can solve MAX BIPARTITE MATCHING by calling our algorithm for MAX FLOW as a subroutine. But on another

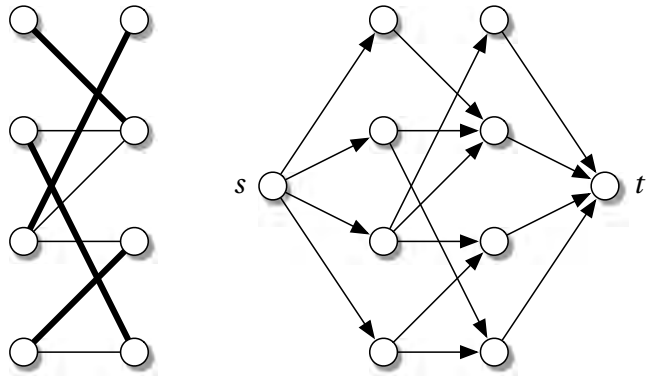


FIGURE 3.23: The reduction from MAX BIPARTITE MATCHING to MAX FLOW. The bipartite graph on the left has a perfect matching, and this corresponds to a flow of value 4 in the directed graph on the right. All edges have capacity 1.

level it says something much deeper—that MAX BIPARTITE MATCHING *is no harder than* MAX FLOW. We write this as an inequality,

$$\text{MAX BIPARTITE MATCHING} \leq \text{MAX FLOW}. \tag{3.16}$$

Consider the fact that, before we saw the polynomial-time algorithm for MAX FLOW, it was not at all obvious that either of these problems are in P. This reduction tells us that if MAX FLOW is in P, then so is MAX BIPARTITE MATCHING. In other words, as soon as we find a polynomial-time algorithm for MAX FLOW we gain one for MAX BIPARTITE MATCHING as well.

Reductions have another important application in computational complexity. Just as a reduction $A \leq B$ shows that *A is at most as hard as B*, it also shows that *B is at least as hard as A*. Thus we get a conditional lower bound on B's complexity as well as a conditional upper bound on A's. In particular, just as $B \in \mathsf{P}$ implies $A \in \mathsf{P}$, it is equally true that $A \notin \mathsf{P}$ implies $B \notin \mathsf{P}$.

If the reader finds the word "reduction" confusing, we sympathize. Saying that A can be reduced to B makes it sound as if B is smaller or simpler than A. In fact, it usually means the reverse: A can be viewed as a special case of B, but B is more general, and therefore harder, than A. Later on, we will put this another way—that B is *expressive enough* to describe the goals and constraints of A.

As we will discuss below, it is generally very hard to prove that a problem is outside P. However, for many problems B we are in the following curious situation: thousands of different problems can be reduced to B, and after decades of effort we have failed to find polynomial-time algorithms for any of them. This gives us very strong evidence that B is outside P. As we will see in the next two chapters, this is exactly the situation for many of the search and optimization problems we care about.

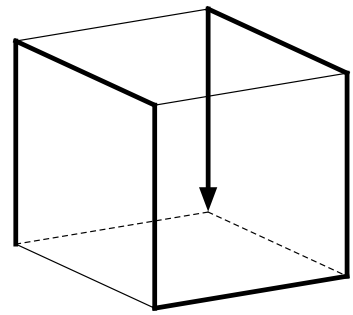


FIGURE 3.24: A Hamiltonian path on the 3-dimensional hypercube.

Problems

Problems worthy of attack
prove their worth by hitting back.

Piet Hein

3.1 Recursive priests are optimal. Prove that the recursive algorithm for the Towers of Hanoi produces the best possible solution, and that this solution is unique: i.e., that it is impossible to solve the puzzle in less than $2^n - 1$ moves, and that there is only one way to do it with this many. Hint: prove this by induction on n. A recursive algorithm deserves a recursive proof.

3.2

3.2 Hamilton visits Hanoi. Find a mapping between the recursive solution of the Towers of Hanoi and a Hamiltonian path on an n-dimensional hypercube (see Figure 3.24). They both have length $2^n - 1$, but what do the vertices and edges of the cube correspond to?

3.3 Hanoi, step by step. Find an *iterative*, rather than recursive, algorithm for the Towers of Hanoi. In other words, find an algorithm that can look at the current position and decide what move to make next. If you like, the algorithm can have a small amount of "memory," such as remembering the most recent move, but it should not maintain a stack of subproblems.

3.4 Our first taste of state space. The state space of the Towers of Hanoi puzzle with 2 disks can be viewed as as a graph with 9 vertices, as shown in Figure 3.25. Describe the structure of this graph for general n, and explain what path the optimal solution corresponds to.

3.5 Four towers. What if there are 4 pegs in the Towers of Hanoi, instead of 3? Here is one possible solution. We will assume that n is a *triangular number*,

$$n_k = 1+2+3+\cdots+k = k(k+1)/2.$$

Then to move n_k disks, recursively move the $n_{k-1} = n_k - k$ smallest disks, i.e., all but the k largest, to one of the four pegs; then, using the other 3 pegs, move the k largest disks using the solution to the 3-peg puzzle; then recursively

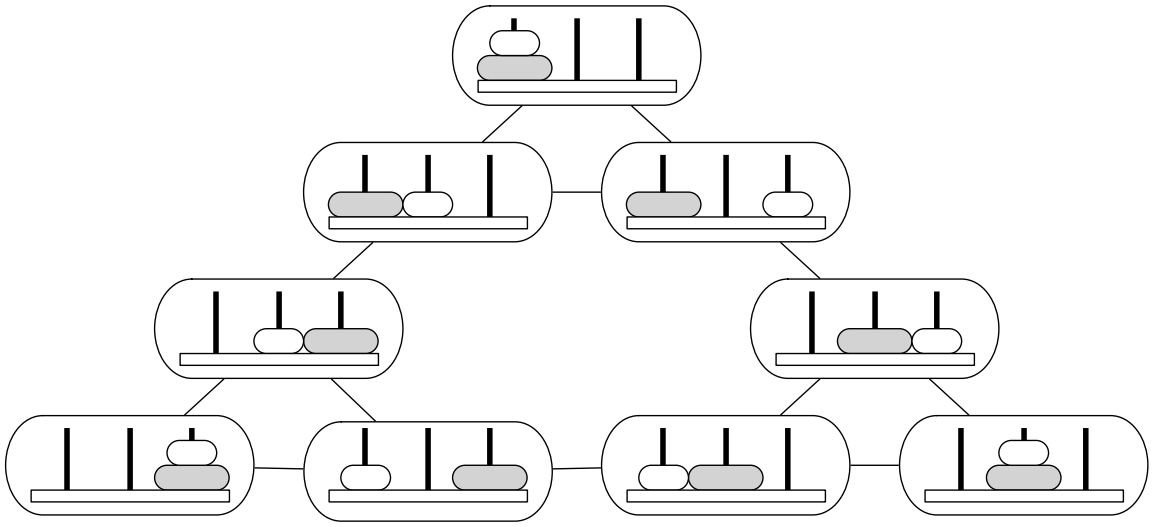


FIGURE 3.25: The state space of all possible positions of the Towers of Hanoi with $n = 2$. Each vertex is a position, and each edge is a legal move.

move the n_{k-1} smallest ones on top of the k largest, at which point we're done. Show that the total number of moves done by this algorithm obeys the equation

$$f(n_k) = 2f(n_{k-1}) + 2^k - 1$$

and that, with the base case $f(n_0) = 0$, the solution is

$$f(n_k) = (k-1)2^k + 1 = \Theta\left(\sqrt{n}\, 2^{\sqrt{2n}}\right).$$

Can you generalize this to 5 or more pegs?

3.6 Pick a card. You walk into a room, and see a row of n cards. Each one has a number x_i written on it, where i ranges from 1 to n. However, initially all the cards are face down. Your goal is to find a *local minimum*: that is, a card i whose number is less than or equal to those of its neighbors, $x_{i-1} \geq x_i \leq x_{i+1}$. The first and last cards can also be local minima, and they only have one neighbor to compare to. There can be many local minima, but you are only responsible for finding one of them.

Obviously you can solve this problem by turning over all n cards, and scanning through them. However, show that you can find such a minimum by turning over only $O(\log n)$ cards.

3.7 The running time of `Mergesort`. Show that the equation for the number of comparisons that `Mergesort` performs on a list of size n,

$$T(n) = 2T(n/2) + n,$$

with the base case $T(1) = 0$, has the exact solution

$$T(n) = n \log_2 n,$$

where we assume that n is a power of 2. Do this first simply by plugging this solution into the equation and checking that it works. Then come up with a more satisfying explanation, based on the number of levels of recursion and the number of comparisons at each level.

We can improve this slightly using the fact that the merge operation takes at most $n-1$ comparisons. Prove that the exact solution to the equation

$$T(n) = 2T(n/2) + n - 1$$

is

$$T(n) = n \log_2 n - n + 1,$$

where as before we assume that n is a power of 2.

3.8 More pieces. Generalizing the previous problem, show that for any positive constants a, b, the equation

$$T(n) = a\,T(n/b) + n^{\log_b a},$$

with the base case $T(1) = 0$, has the exact solution

$$T(n) = n^{\log_b a} \log_b n,$$

where we assume that n is a power of b.

3.9 Integrating quicksort. Let's solve the equation (3.4) for the average running time of `Quicksort`. First, show that if n is large, we can replace the sum by an integral and obtain

$$T(n) = n + \frac{2}{n} \int_0^n T(x)\,\mathrm{d}x. \tag{3.17}$$

Since we expect the running time to be $\Theta(n \log n)$, let's make a guess that

$$T(n) = An \ln n$$

for some constant A. Note that we use the natural logarithm since it is convenient for calculus. Substitute this guess into (3.17) and show that $A = 2$, so that $T(n) \approx 2n \ln n$ as stated in the text.

3.10 Better pivots. In general, the running time $T(n)$ of `Quicksort` obeys the equation

$$T(n) = n + \int_0^n P(r)(T(r) + T(n-r))\,\mathrm{d}r,$$

where $P(r)$ is the probability that the pivot has rank r. For instance, (3.4) is the special case where the pivot is a random element, so $P(r) = 1/n$ for all r.

One common improvement to `Quicksort` is to choose *three* random elements, and let the pivot be the median of these three. Calculate the probability distribution $P(r)$ of the pivot's rank in this case. Solve this equation in the limit of large n by converting it to an integral as in Problem 3.9, and again try a solution of the form $T(n) = An \ln n$. How much does this technique improve the constant A? What happens to A if the pivot is the median of 5 elements, 7 elements, and so on?

3.11 Partway to the median. Imagine that the pivot in `Quicksort` always divides the list into two sublists of size γn and $(1-\gamma)n$, so that the number of comparisons obeys the equation

$$T(n) = T(\gamma n) + T((1-\gamma)n) + n.$$

Show that $T(n) = A(\gamma)\, n \ln n$, and determine the constant $A(\gamma)$. How does $A(\gamma)$ behave when γ is close to 0 or to 1?

3.12 Factoring with factorials. Consider the following problem:

MODULAR FACTORIAL
Input: Two n-digit integers x and y
Output: $x! \bmod y$

This sounds a lot like MODULAR EXPONENTIATION, and it is tempting to think that MODULAR FACTORIAL is in P as well. However, show that

$$\text{FACTORING} \leq \text{MODULAR FACTORIAL},$$

in the sense that we can solve FACTORING by calling a subroutine for MODULAR FACTORIAL a polynomial number of times. Thus if MODULAR FACTORIAL is in P, then FACTORING is in P as well. Hint: first note that y is prime if and only if $\gcd(x!, y) = 1$ for all $x < y$.

3.13 Gamma function identities. (A follow-up to the previous problem.) There is a natural generalization of the factorial to noninteger values, called the *Gamma function*:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \, dt. \tag{3.18}$$

We have $\Gamma(x+1) = x\Gamma(x)$, and in particular $\Gamma(x) = (x-1)!$ if x is an integer. But $\Gamma(1/2)$, for instance, is $\sqrt{\pi}$.

Now consider the following formula, which *almost* gives a divide-and-conquer algorithm for $\Gamma(x)$:

$$\Gamma(x)\Gamma(x+1/2) = 2^{1-2x}\sqrt{\pi}\,\Gamma(2x). \tag{3.19}$$

Suppose there were an identity of the form

$$\Gamma(2x) = f(x)\Gamma(x)^2$$

for some function f such that, when x is an integer, $f(x) \bmod y$ can be computed in polynomial time for n-digit integers x and y. Show that then MODULAR FACTORIAL, and therefore FACTORING, would be in P.


3.12

3.14 FFTs for smooth numbers. Let's generalize the running time of the Fast Fourier Transform to values of n other than powers of 2. If n has a factor p and we divide the list into p sublists of size n/p, argue that the running time obeys

$$T(n) = pT(n/p) + pn,$$

where for the purposes of elegance we omit the Θ on the right-hand-side. Let $\{p_i\}$ be the prime factorization of n, with each prime repeated the appropriate number of times: i.e., $n = \prod_i p_i$. Taking the base case $T(p) = p^2$ for prime p, show that the running time is

$$T(n) = n\sum_i p_i.$$

Now, a number is called *q-smooth* if all of its prime factors are smaller than q. Show that if n is q-smooth for some constant q, the running time is $\Theta(n \log n)$.

3.15 Convolutions and polynomials. Given two functions f, g, their *convolution* $f \star g$ is defined as

$$(f \star g)(t) = \sum_s f(s)\, g(t-s).$$

Convolutions are useful in signal processing, random walks, and fast algorithms for multiplying integers and polynomials. For instance, suppose that we have two polynomials whose tth coefficients are given by $f(t)$ and $g(t)$ respectively:

$$P(z)=\sum_t f(t)z^t\,,\quad Q(z)=\sum_t g(t)z^t\,.$$

Then show that the coefficients of their product are given by the convolution of f and g,

$$P(z)Q(z)=\sum_t (f\star g)(t)z^t\,.$$

Now, as in Problem 2.14, let's represent integers as polynomials, whose coefficients are the integers' digits. For instance, we can write $729=P(10)$ where $P(z)=7z^2+2z+9$. Show that, except for some carrying, the product of two integers is given by the product of the corresponding polynomials, and so in turn by the convolution of the corresponding functions.

3.16 Convolving in Fourier space. Continuing from the previous problem, suppose we have two functions $f(t), g(t)$ defined on the integers mod n. Their convolution is

$$(f\star g)(t)=\sum_{s=0}^{n-1} f(s)\,g(t-s),$$

where $t-s$ is now evaluated mod n. How long does it take to calculate $f\star g$ from f and g? Evaluating the sum directly for each t takes $O(n^2)$ time, but we can do much better by using the Fourier transform.

Show that the Fourier transform of the convolution is the product of the Fourier transforms. More precisely, show that for each frequency k from 0 to $n-1$,

$$\widetilde{(f\star g)}(k)=\tilde f(k)\,\tilde g(k).$$

Thus we can find $f\star g$ by Fourier transforming f and g, multiplying them together, and then inverse Fourier transforming the result.

Conclude that if we use the Fast Fourier Transform, we can convolve two functions defined on the integers mod n in $O(n\log n)$ time. If necessary, we can "pad" the two functions out, repeating one and adding zeros to the other, so that n is a power of 2. This is the heart of the Schönhage–Strassen algorithm for integer multiplication discussed in Note 2.5. It is also at the heart of virtually all modern signal processing.

3.17 FFTs for primes. The Fast Fourier Transform described in the text works well if n has many small factors. But what do we do if n is prime?

First, note that when the frequency is zero, the Fourier transform is just proportional to the average, $\tilde f(0)=(1/\sqrt{n})\sum_t f(t)$, which we can easily calculate in $O(n)$ time. So, we focus our attention on the case where $k\neq 0$, and write (3.6) as follows:

$$\tilde f(k)=\frac{1}{\sqrt{n}}\sum_{t=0}^{n-1} f(t)\,\mathrm{e}^{-2\imath\pi kt/n}=\frac{1}{\sqrt{n}}\left(f(0)+S(k)\right)$$

where

$$S(k)=\sum_{t=1}^{n-1} f(t)\,\mathrm{e}^{-2\imath\pi kt/n}\,.$$

Now, for any prime n, there is an integer a whose powers $1,a,a^2,\ldots,a^{n-2}$, taken mod n, range over every possible t from 1 to $n-1$. Such an a is called a *primitive root*; algebraically, a is a generator of the multiplicative group $\mathbb{Z}_n^*$ (see Appendix A.7). By writing $k=a^\ell$ and $t=a^m$ and rearranging, show that S is the convolution of two functions, each of which is defined on the integers mod $n-1$. Since in the previous problem we showed how to convolve two functions in time $O(n\log n)$, this gives a Fast Fourier Transform even when n is prime.

3.18 Dynamic Fibonacci. Suppose I wish to calculate the nth Fibonacci number recursively, using the equation $F_\ell = F_{\ell-1} + F_{\ell-2}$ and the base case $F_1 = F_2 = 1$. Show that if I do this without remembering values of F_ℓ that I calculated before, then the number of recursive function calls is exponentially large. In fact, it is essentially F_ℓ itself. Then show that if I memorize previous values, calculating F_ℓ takes only $O(\ell)$ function calls. (Of course, this is exponential in the number of digits of ℓ.)

3.19 Divide-and-conquer Fibonacci. Let F_ℓ denote the ℓth Fibonacci number. Starting from the equation $F_\ell = F_{\ell-1} + F_{\ell-2}$ and the base case $F_1 = F_2 = 1$, prove that for any ℓ and any $m \geq 1$ we have

$$F_\ell = F_{m+1}\, F_{\ell-m} + F_m\, F_{\ell-m-1}\,. \tag{3.20}$$

In particular, prove the following:

$$\begin{aligned} F_{2\ell} &= F_\ell^2 + 2F_\ell\, F_{\ell-1} \\ F_{2\ell+1} &= F_{\ell+1}^2 + F_\ell^2\,. \end{aligned} \tag{3.21}$$

Then show that if ℓ and p are n-bit numbers, we can calculate $F_\ell \bmod p$ in poly(n) time. Hint: using (3.21) naively gives a running time that is polynomial in ℓ, but not in $n = \log_2 \ell$.

3.20 Getting through the paragraph. Show that the TYPESETTING problem can be described as SHORTEST PATH on a weighted graph with a polynomial number of vertices. What do the vertices of this graph represent, and what are the weights of the edges between them?

3.21 Alignments are paths. At first, calculating the minimum edit distance $d(s,t)$ seems like finding a shortest path in an exponentially large graph, namely the graph of all strings of length $n = \max(|s|,|t|)$ where two strings are neighbors if they differ by a single insertion, deletion, or mutation. However, it can be reduced to SHORTEST PATH on a graph of size $O(n^2)$ in the following way. For each $0 \leq i \leq |s|$ and $0 \leq j \leq |t|$, let v_{ij} be a vertex corresponding to a point in the alignment that has already accounted for the first i symbols of s and the first j symbols of t. Show how to assign weights to the edges between these vertices so that $d(s,t)$ is the length of the shortest path from $v_{0,0}$ to $v_{|s|,|t|}$.

3.22 Increasing subsequences. Given a sequence of integers $s_1, s_2, \ldots, s_n$, an *increasing subsequence* of length k is a series of indices $i_1 < i_2 < \cdots < i_k$ such that $s_{i_1} < s_{i_2} < \cdots < s_{i_k}$. For instance, the sequence

$$6, 3, 4, 8, 1, 5, 7, 2, 9$$

has an increasing subsequence of length 5, namely

$$3, 4, 5, 7, 9\,.$$

Even though there are an exponential number of possible subsequences, show that the problem of finding the longest one can be solved in polynomial time.

3.23 A pigeon ascending. Now prove that any sequence $s_1, \ldots, s_n$ of distinct integers has either an increasing subsequence, or a decreasing subsequence, of length at least $\lceil\sqrt{n}\rceil$. Hint: for each i, let a_i and b_i, respectively, be the length of the longest increasing and decreasing subsequence ending with s_i. Then show that for every $i \neq j$, either $a_i \neq a_j$ or $b_i \neq b_j$, and use the pigeonhole principle from Problem 1.2.

3.13

3.24 Multiplying matrices mellifluously. Suppose that I have a sequence of matrices, $M^{(1)}, \ldots, M^{(n)}$, where each $M^{(t)}$ is an $a_t \times b_t$ matrix and where $b_t = a_{t+1}$ for all $1 \leq t < n$. I wish to calculate their matrix product $M = \prod_{t=1}^{n} M^{(t)}$, i.e., the matrix such that

$$M_{ij} = \sum_{\ell_1=1}^{b_1} \sum_{\ell_2=1}^{b_2} \cdots \sum_{\ell_{n-1}}^{b_{n-1}} M^{(1)}_{i,\ell_1} M^{(2)}_{\ell_1,\ell_2} \cdots M^{(n)}_{\ell_{n-1},j}\,.$$

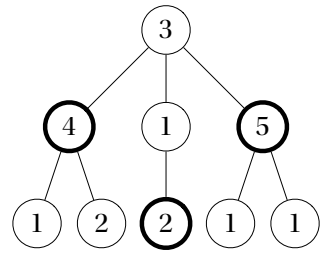


FIGURE 3.26: A tree with vertex weights and its maximum-weight independent set.

However, the number of multiplications I need to do this depends on how I parenthesize this product. For instance, if $M^{(1)}$ is a $1 \times k$ row vector, $M^{(2)}$ and $M^{(3)}$ are $k \times k$ matrices, and $M^{(4)}$ is a $k \times 1$ column vector, then

$$M = \left(M^{(1)} \cdot M^{(2)}\right) \cdot \left(M^{(3)} \cdot M^{(4)}\right)$$

takes $k^2 + k^2 + k$ multiplications, while

$$M = M^{(1)} \cdot \left(\left(M^{(2)} \cdot M^{(3)}\right) \cdot M^{(4)}\right)$$

takes $k^3 + k^2 + k$ multiplications. Show how to find the optimal parenthesization, i.e., the one with the smallest number of multiplications, in polynomial time.

3.25 Prune and conquer. Suppose I have a graph $G = (V, E)$ where each vertex v has a weight $w(v)$. An *independent set* is a subset S of V such that no two vertices in S have an edge between them. Consider the following problem:

MAX-WEIGHT INDEPENDENT SET
Input: A graph $G = (V, E)$ with vertex weights $w(v)$
Question: What is the independent set with the largest total weight?

Use dynamic programming to show that, in the special case where G is a tree, MAX-WEIGHT INDEPENDENT SET is in P.

Hint: once you decide whether or not to include the root of the tree in the set S, how does the remainder of the problem break up into pieces? Consider the example in Figure 3.26. Note that a greedy strategy doesn't work.

Many problems that are easy for trees seem to be very hard for general graphs, and we will see in Chapter 5 that MAX-WEIGHT INDEPENDENT SET is one of these. Why does dynamic programming not work for general graphs?

3.26 Prune and conquer again. Suppose I have a graph $G = (V, E)$ where each edge e has a weight $w(e)$. Recall that a *matching* is a subset $M \subseteq E$ such that no two edges in M share an endpoint. Consider the weighted version of MAX MATCHING, illustrated in Figure 3.27:

MAX-WEIGHT MATCHING
Input: A graph $G = (V, E)$ with edge weights $w(e)$
Question: What is the partial matching with the largest total weight?

Using dynamic programming, give a polynomial-time algorithm for this problem in the case where G is a tree, similar to that for Problem 3.25.