Algorithms Illuminated

Part 4: Algorithms for NP-Hard Problems

Tim Roughgarden

First Edition

Cover image:
Norman Zammitt, *Blue Burning*, 1982
acrylic on canvas; 84 x 168 1/2 in. (213.36 x 427.99 cm)
San Francisco Museum of Modern Art, Gift of Judge and Mrs. William J. Lasarow
© Estate of Norman Zammitt
Photograph: Katherine Du Tiel

ISBN: 978-0-9992829-6-0 (Paperback)
ISBN: 978-0-9992829-7-7 (ebook)

Library of Congress Control Number: 2017914282

Soundlikeyourself Publishing, LLC
New York, NY
`soundlikeyourselfpublishing@gmail.com`
`www.algorithmsilluminated.org`

Contents

Preface

This book is the fourth in a series based on my online algorithms courses that have been running regularly since 2012, which in turn are based on an undergraduate course that I taught many times at Stanford University. *Part 4* assumes at least some familiarity with asymptotic analysis and big-O notation, graph search and shortest-path algorithms, greedy algorithms, and dynamic programming (all covered in *Parts 1–3*).

What We'll Cover in This Book

Algorithms Illuminated, Part 4 is all about NP-hard problems and what to do about them.

Algorithmic tools for tackling NP-hard problems. Many real-world problems are "NP-hard" and appear unsolvable by the types of always-correct and always-fast algorithms that have starred in the first three parts of this book series. When an NP-hard problem shows up in your own work, you must compromise on either correctness or speed. We'll see techniques old (like greedy algorithms) and new (like local search) for devising fast heuristic algorithms that are "approximately correct," with applications to scheduling, influence maximization in social networks, and the traveling salesman problem. We'll also cover techniques old (like dynamic programming) and new (like MIP and SAT solvers) for developing correct algorithms that improve dramatically on exhaustive search; applications here include the traveling salesman problem (again), finding signaling pathways in biological networks, and television station repacking in a recent and high-stakes spectrum auction in the United States.

Recognizing NP-hard problems. This book will also train you to quickly recognize an NP-hard problem so that you don't inadvertently waste time trying to design a too-good-to-be-true algorithm for it. You'll acquire familiarity with many famous and basic NP-hard problems, ranging from satisfiability to graph coloring to the Hamiltonian path problem. Through practice, you'll learn the tricks of the trade in proving problems NP-hard via reductions.

For a more detailed look into the book's contents, check out the "Upshot" sections that conclude each chapter and highlight the most important points. The "Field Guide to Algorithm Design" on page 236 provides a bird's-eye view of how the topics of this book fit into the bigger algorithmic picture.

The starred sections of the book are the most advanced ones. The time-constrained reader can skip these sections on a first reading without any loss of continuity.

Topics covered in the first three parts. *Algorithms Illuminated, Part 1* covers asymptotic notation (big-O notation and its close cousins), divide-and-conquer algorithms and the master method, randomized QuickSort and its analysis, and linear-time selection algorithms. *Part 2* is about data structures (heaps, balanced search trees, hash tables, bloom filters), graph primitives (breadth- and depth-first search, connectivity, shortest paths), and their applications (ranging from deduplication to social network analysis). *Part 3* focuses on greedy algorithms (scheduling, minimum spanning trees, clustering, Huffman codes) and dynamic programming (knapsack, sequence alignment, shortest paths, optimal search trees).

Skills You'll Learn From This Book Series

Mastering algorithms takes time and effort. Why bother?

Become a better programmer. You'll learn several blazingly fast subroutines for processing data as well as several useful data structures for organizing data that you can deploy directly in your own programs. Implementing and using these algorithms will stretch and improve your programming skills. You'll also learn general algorithm design paradigms that are relevant to many different problems across different domains, as well as tools for predicting the performance of such algorithms. These "algorithmic design patterns" can help you come up with new algorithms for problems that arise in your own work.

Sharpen your analytical skills. You'll get lots of practice describing and reasoning about algorithms. Through mathematical analysis, you'll gain a deep understanding of the specific algorithms and data structures that these books cover. You'll acquire facility with several mathematical techniques that are broadly useful for analyzing algorithms.

Think algorithmically. After you learn about algorithms, you'll start seeing them everywhere, whether you're riding an elevator, watching a flock of birds, managing your investment portfolio, or even watching an infant learn. Algorithmic thinking is increasingly useful and prevalent in disciplines outside of computer science, including biology, statistics, and economics.

Literacy with computer science's greatest hits. Studying algorithms can feel like watching a highlight reel of many of the greatest hits from the last sixty years of computer science. No longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra's algorithm. After reading these books, you'll know exactly what they mean.

Ace your technical interviews. Over the years, countless students have regaled me with stories about how mastering the concepts in these books enabled them to ace every technical interview question they were ever asked.

How These Books Are Different

This series of books has only one goal: *to teach the basics of algorithms in the most accessible way possible.* Think of them as a transcript of what an expert algorithms tutor would say to you over a series of one-on-one lessons.

There are a number of excellent more traditional and encyclopedic textbooks about algorithms, any of which usefully complement this book series with additional details, problems, and topics. I encourage you to explore and find your own favorites. There are also several books that, unlike these books, cater to programmers looking for ready-made algorithm implementations in a specific programming language. Many such implementations are freely available on the Web as well.

Who Are You?

The whole point of these books and the online courses upon which they are based is to be as widely and easily accessible as possible. People of all ages, backgrounds, and walks of life are well represented in my online courses, and there are large numbers of students (high-school, college, etc.), software engineers (both current and aspiring), scientists, and professionals hailing from all corners of the world.

This book is not an introduction to programming, and ideally you've acquired basic programming skills in a standard language (like Java, Python, C, Scala, Haskell, etc.). If you need to beef up your programming skills, there are several outstanding free online courses that teach basic programming.

We also use mathematical analysis as needed to understand how and why algorithms really work. The freely available book *Mathematics for Computer Science*, by Eric Lehman, F. Thomson Leighton, and Albert R. Meyer, is an excellent and entertaining refresher on mathematical notation (like $\sum$ and $\forall$), the basics of proofs (induction, contradiction, etc.), discrete probability, and much more.

Additional Resources

These books are based on online courses that are currently running on the Coursera and EdX platforms. I've made several resources available to help you replicate as much of the online course experience as you like.

Videos. If you're more in the mood to watch and listen than to read, check out the YouTube video playlists available at `www.algorithmsilluminated.org`. These videos cover all the topics in this book series, as well as additional advanced topics. I hope they exude a contagious enthusiasm for algorithms that, alas, is impossible to replicate fully on the printed page.

Quizzes. How can you know if you're truly absorbing the concepts in this book? Quizzes with solutions and explanations are scattered throughout the text; when you encounter one, I encourage you to pause and think about the answer before reading on.

End-of-chapter problems. At the end of each chapter, you'll find several relatively straightforward questions that test your understanding, followed by harder and more open-ended challenge problems. Hints or solutions to all of these problems (as indicated by an "*(H)*" or "*(S)*," respectively) are included at the end of the book. Readers can interact with me and each other about the end-of-chapter problems through the book's discussion forum (see below).

Programming problems. Several of the chapters conclude with suggested programming projects whose goal is to help you develop a detailed understanding of an algorithm by creating your own working implementation of it. Data sets, along with test cases and their solutions, can be found at `www.algorithmsilluminated.org`.

Discussion forums. A big reason for the success of online courses is the opportunities they provide for participants to help each other understand the course material and debug programs through discussion forums. Readers of these books have the same opportunity via the forums available at `www.algorithmsilluminated.org`.

Acknowledgments

These books would not exist without the passion and hunger supplied by the hundreds of thousands of participants in my algorithms courses over the years. I am particularly grateful to those who supplied detailed feedback on an earlier draft of this book: Tonya Blust, Yuan Cao, Leslie Damon, Tyler Dae Devlin, Roman Gafiteanu, Blanca Huergo, Jim Humelsine, Tim Kearns, Vladimir Kokshenev, Bayram Kuliyev, Clayton Wong, Lexin Ye, and Daniel Zingaro. Thanks also to several experts who provided technical advice: Amir Abboud, Vincent Conitzer, Christian Kroer, Aviad Rubinstein, and Ilya Segal.

I always appreciate suggestions and corrections from readers. These are best communicated through the discussion forums mentioned above.

Tim Roughgarden
New York, NY
June 2020

Chapter 19

What Is NP-Hardness?

Introductory books on algorithms, including *Parts 1–3* of this series, suffer from selection bias. They focus on computational problems that are solvable by clever, fast algorithms—after all, what's more fun and empowering to learn than an ingenious algorithmic short-cut? The good news is that many fundamental and practically relevant problems fall into this category: sorting, graph search, shortest paths, Huffman codes, minimum spanning trees, sequence alignment, and so on. But it would be fraudulent to teach you only this cherry-picked collection of problems while ignoring the spectre of computational intractability that haunts the serious algorithm designer or programmer. Sadly, there are many important computational problems, including ones likely to show up in your own projects, for which no fast algorithms are known. Even worse, we can't expect any future algorithmic breakthroughs for these problems, as they are widely believed to be intrinsically difficult and unsolvable by any fast algorithm.

Newly aware of this stark reality, two questions immediately come to mind. First, how can you recognize such hard problems when they appear in your own work, so that you can adjust your expectations accordingly and avoid wasting time looking for a too-good-to-be-true algorithm? Second, when such a problem is important to your application, how should you revise your ambitions, and what algorithmic tools can you apply to achieve them? This book will equip you with thorough answers to both questions.

19.1 MST vs. TSP: An Algorithmic Mystery

Hard computational problems can look a lot like easy ones, and telling them apart requires a trained eye. To set the stage, let's rendezvous with a familiar friend (the minimum spanning tree problem) and meet its more demanding cousin (the traveling salesman problem).

19.1.1 The Minimum Spanning Tree Problem

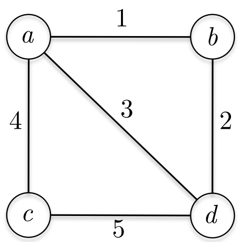
One famous computational problem solvable by a blazingly fast algorithm is the *minimum spanning tree (MST)* problem (covered in Chapter 15 of *Part 3*).[1]

Problem: Minimum Spanning Tree (MST)

Input: A connected undirected graph $G = (V, E)$ and a real-valued cost c_e for each edge $e \in E$.

Output: A spanning tree $T \subseteq E$ of G with the minimum-possible sum $\sum_{e \in T} c_e$ of edge costs.

Recall that a graph $G = (V, E)$ is *connected* if, for every pair $v, w \in V$ of vertices, the graph contains a path from v to w. A *spanning tree* of G is a subset $T \subseteq E$ of edges such that the subgraph (V, T) is both connected and acyclic. For example, in the graph



the minimum spanning tree comprises the edges (a, b), (b, d), and (a, c), for an overall cost of 7.

A graph can have an exponential number of spanning trees, so exhaustive search is out of the question for all but the smallest graphs.[2] But the MST problem *can* be solved by clever fast algorithms,

[1]To review, a *graph* $G = (V, E)$ has two ingredients: a set V of *vertices* and a set E of *edges*. In an *undirected* graph, each edge $e \in E$ corresponds to an unordered pair $\{v, w\}$ of vertices (written as $e = (v, w)$ or $e = (w, v)$). In a *directed* graph, each edge (v, w) is an ordered pair, with the edge directed from v to w. The numbers $|V|$ and $|E|$ of vertices and edges are usually denoted by n and m, respectively.

[2]For example, *Cayley's formula* is a famous result from combinatorics stating that the n-vertex complete graph (in which all the $\binom{n}{2}$ possible edges are present) has exactly n^{n-2} different spanning trees. This is bigger than the estimated number of atoms in the known universe when $n \geq 50$.

such as Prim's and Kruskal's algorithms. Deploying appropriate data structures (heaps and union-find, respectively), both algorithms have blazingly fast implementations, with a running time of $O((m + n)\log n)$, where m and n are the number of edges and vertices of the input graph, respectively.

19.1.2 The Traveling Salesman Problem

Another famous problem, absent from *Parts 1–3* but prominent in this book, is the *traveling salesman problem (TSP)*. Its definition is almost the same as that of the MST problem, except with *tours*—simple cycles that span all vertices—playing the role of spanning trees.

Problem: Traveling Salesman Problem (TSP)

Input: A complete undirected graph $G = (V, E)$ and a real-valued cost c_e for each edge $e \in E$.[3]

Output: A tour $T \subseteq E$ of G with the minimum-possible sum $\sum_{e \in T} c_e$ of edge costs.

Formally, a *tour* is a cycle that visits every vertex exactly once (with two edges incident to each vertex).

Quiz 19.1

In an instance $G = (V, E)$ of the TSP with $n \geq 3$ vertices, how many distinct tours $T \subseteq E$ are there? (In the answers below, $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ denotes the factorial function.)

a) 2^n

b) $\frac{1}{2}(n-1)!$

[3]In a *complete* graph, all $\binom{n}{2}$ possible edges are present. The assumption that the graph is complete is without loss of generality, as an arbitrary input graph can be harmlessly turned into a complete graph by adding in all the missing edges and giving them very high costs.

c) $(n-1)!$

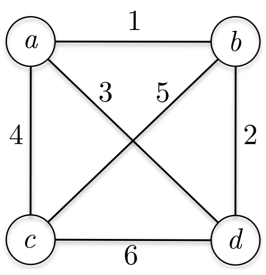
d) $n!$

(See Section 19.1.4 for the solution and discussion.)

If all else fails, the TSP can be solved by exhaustively enumerating all of the (finitely many) tours and remembering the best one. Try exhaustive search out on a small example.

Quiz 19.2

What is the minimum sum of edge costs of a tour of the following graph? (Each edge is labeled with its cost.)



a) 12

b) 13

c) 14

d) 15

(See Section 19.1.4 for the solution and discussion.)

The TSP can be feasibly solved by exhaustive search for only the smallest of instances. *Can we do better?* Could there be, analogous to the MST problem, an algorithm that magically homes in on the minimum-cost needle in the exponential-size haystack of traveling salesman tours? Despite the superficial similarity of the statements of the two problems, the TSP appears to be far more difficult to solve than the MST problem.

19.1.3 Trying and Failing to Solve the TSP

I could tell you a cheesy story about, um, a traveling salesman, but this would do a disservice to the TSP, which is actually quite fundamental. Whenever you have a bunch of tasks to complete in a sequence, with the cost or time for carrying out a task dependent on the preceding task, you're talking about the TSP in disguise.

For example, tasks could represent cars to be assembled in a factory, with the time required to assemble a car equal to a fixed cost (for assembly) plus a setup cost that depends on how different the factory configurations are for this and the previous car. Assembling all the cars as quickly as possible boils down to minimizing the sum of the setup costs, which is exactly the TSP.

For a very different application, imagine that you've collected a bunch of overlapping fragments of a genome and would like to reverse engineer their most plausible ordering. Given a "plausibility measure" that assigns a cost to each fragment pair (for example, derived from the length of their longest common substring), this ordering problem also boils down to the TSP.[4]

Seduced by the practical applications and aesthetic appeal of the TSP, many of the greatest minds in optimization have, since at least the early 1950s, devoted a tremendous amount of effort and computation to solving large-scale instances of the TSP.[5] Despite the decades and intellectual firepower involved:

Fact

As of this writing (in 2020), there is no known fast algorithm for the TSP.

What do we mean by a "fast" algorithm? Back in *Part 1*, we agreed that:

[4]Both applications are arguably better modeled as traveling salesman *path* problems, in which the goal is to compute a minimum-cost cycle-free path that visits every vertex (without going back to the starting vertex). Any algorithm solving the TSP can be easily converted into one solving the path version of the problem, and vice versa (Problem 19.7).

[5]Readers curious about the history or additional applications of the TSP should check out the first four chapters of the book *The Traveling Salesman Problem: A Computational Study*, by David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook (Princeton University Press, 2006).

> *A "fast algorithm" is an algorithm whose worst-case running time grows slowly with the input size.*

And what do we mean by "grows slowly"? For much of this book series, the holy grail has been algorithms that run in linear or almost-linear time. Forget about such blazingly fast algorithms—for the TSP, no one even knows of an algorithm that always runs in $O(n^{100})$ time on n-vertex instances, or even $O(n^{10000})$ time.

There are two competing explanations for the dismal state-of-the-art: (i) there is a fast algorithm for the TSP but no one's been smart enough to find it yet; or (ii) no such algorithm exists. We do not know which explanation is correct, though most experts believe in the second one.

Speculation

No fast algorithm for the TSP exists.

As early as 1967, Jack Edmonds wrote:

> I conjecture that there is no good algorithm for the traveling saleman [sic] problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know.[6]

Unfortunately, the curse of intractability is not confined to the TSP. We'll see that many other practically relevant problems are similarly afflicted.

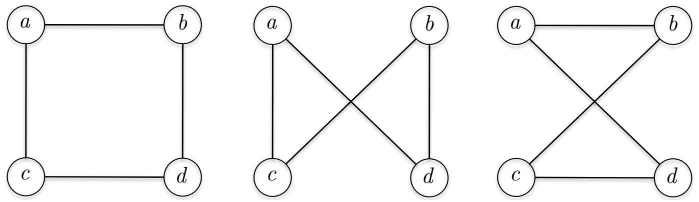
19.1.4 Solutions to Quizzes 19.1–19.2

Solution to Quiz 19.1

Correct answer: (b). There is an intuitive correspondence between vertex orderings (of which there are $n!$) and tours (which visit the vertices once each, in some order), so answer (d) is a natural guess. However, this correspondence counts each tour in $2n$ different ways:

[6]From the paper "Optimum Branchings," by Jack Edmonds (*Journal of Research of the National Bureau of Standards, Series B*, 1967). By a "good" algorithm, Edmonds means an algorithm with a running time bounded above by some polynomial function of the input size.

once for each of the n choices of the initial vertex and once for each of the two directions of traversing the tour. Thus, the total number of tours is $n!/2n = \frac{1}{2}(n-1)!$. For example, with $n = 4$, there are three distinct tours:



Solution to Quiz 19.2

Correct answer: (b). We can enumerate tours by starting with the vertex a and trying all six possible orderings of the other three vertices, with the understanding that the tour finishes by traveling from the last vertex back to a. (Actually, this enumeration counts each tour twice, once in each direction.) The results:

Vertex Ordering	Cost of Corresponding Tour
a, b, c, d or a, d, c, b	15
a, b, d, c or a, c, d, b	13
a, c, b, d or a, d, b, c	14

The shortest tour is the second one, with a total cost of 13.

19.2 Possible Levels of Expertise

Some computational problems are easier than others. The point of the theory of NP-hardness is to classify, in a precise sense, problems as either "computationally easy" (like the MST problem) or "computationally difficult" (like the TSP). This book is aimed both at readers looking for a white-belt primer on the topic and at those pursuing black-belt expertise. This section offers guidance on how to approach the rest of the book, as a function of your goals and constraints.

What are your current and desired levels of expertise in recognizing and tackling NP-hard problems?[7]

[7]What's up with the term "NP"? See Section 19.6.

Level 0: "What's an NP-hard problem?"

Level 0 is total ignorance—you've never heard of NP-hardness and are unaware that many practically relevant computational problems are widely believed to be unsolvable by any fast algorithm. If I've done my job, this book should be accessible even to level-0 readers.

Level 1: "Oh, the problem is NP-hard? I guess we should either reformulate the problem, scale down our ambitions, or invest a lot more resources into solving it."

Level 1 represents cocktail-party-level awareness and at least an informal understanding of what NP-hardness means.[8] For example, are you managing a software project with an algorithmic or optimization component? If so, you should acquire at least level-1 knowledge, in case one of your team members bumps into an NP-hard problem and wants to discuss the possible next steps. To raise your level to 1, study Sections 19.3, 19.4, and 19.6.

Level 2: "Oh, the problem is NP-hard? Give me a chance to apply my algorithmic expertise and see how far I can get."

The biggest marginal empowerment for software engineers comes from reaching level 2, and acquiring a rich toolbox for developing practically useful algorithms for solving or approximating NP-hard problems. Serious programmers should shoot for this level (or above). Happily, all the algorithmic paradigms that we developed for polynomial-time solvable problems in *Parts 1–3* are also useful for making headway on NP-hard problems. The goal of Chapters 20 and 21 is to bring you up to level 2; see also Section 19.4 for an overview and Chapter 24 for a detailed case study of the level-2 toolbox in action in a high-stakes application.

Level 3: "Tell me about your computational problem. [. . . listens carefully . . .] My condolences, your problem is NP-hard."

At level 3, you can quickly recognize NP-hard problems when they arise in practice (at which point you can switch to applying your level-2 skills). You know several famous NP-hard problems and also

[8]Speaking, as always, about sufficiently nerdy cocktail parties!

how to prove that additional problems are NP-hard. Specialists in algorithms should master these skills. For example, I frequently draw on level-3 knowledge when advising colleagues, students, or engineers in industry on algorithmic problems. Chapter 22 provides a boot camp for upping your game to level 3; see also Section 19.5 for an overview.

$\boxed{\text{Level 4:}}$ "Allow me to explain the P $\neq$ NP conjecture to you on this whiteboard."

Level 4, the most advanced level, is for budding theoreticians and anyone seeking a rigorous mathematical understanding of NP-hardness and the P vs. NP question. If that qualifier doesn't scare you off, the optional Chapter 23 is for you.

19.3 Easy and Hard Problems

An oversimplification of the "easy vs. hard" dichotomy proposed by the theory of NP-hardness is:

easy	$\leftrightarrow$	can be solved with a polynomial-time algorithm;
hard	$\leftrightarrow$	requires exponential time in the worst case.

This summary of NP-hardness overlooks several important subtleties (see Section 19.3.9). But ten years from now, if you remember only a few words about the meaning of NP-hardness, these are good ones.

19.3.1 Polynomial-Time Algorithms

To segue into the definition of an "easy" problem, let's recap the running times of some famous algorithms that you may have seen (for example, in *Parts 1–3*):

Problem	Algorithm	Running Time
Sorting	`MergeSort`	$O(n \log n)$
Strong Components	`Kosaraju`	$O(m+n)$
Shortest Paths	`Dijkstra`	$O((m+n)\log n)$
MST	`Kruskal`	$O((m+n)\log n)$
Sequence Alignment	`NW`	$O(mn)$
All-Pairs Shortest Paths	`Floyd-Warshall`	$O(n^3)$

The exact meaning of n and m is problem-specific, but in all cases they are closely related to the input size.[9] The key takeaway from this table is that, while the running times of these algorithms vary, *all of them are bounded above by some polynomial function of the input size.* In general:

Polynomial-Time Algorithms

A *polynomial-time algorithm* is an algorithm with worst-case running time $O(n^d)$, where n denotes the input size and d is a constant (independent of n).

The six algorithms listed at the beginning of this section are all polynomial-time algorithms (with reasonably small exponents d).[10] Do all natural algorithms run in polynomial time? No. For example, for many problems, exhaustive search runs in time exponential in the input size (as noted in footnote 2 for the MST problem). There's something special about the clever polynomial-time algorithms that we've studied so far.

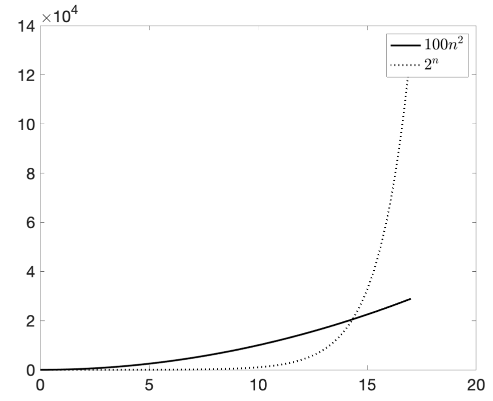
19.3.2 Polynomial vs. Exponential Time

Don't forget that any exponential function eventually grows much faster than any polynomial function. There's a huge difference between typical polynomial and exponential running times, even for very small instances. The plot at the top of the next page (of the polynomial function $100n^2$ versus the exponential function 2^n) is representative.

Moore's law asserts that the computing power available for a given price doubles every 1–2 years. Does this mean that the difference between polynomial-time and exponential-time algorithms will disappear over time? Actually, the exact opposite is true! Our computational ambitions grow with our computational power, and as time goes on we consider increasingly large input sizes and suffer an increasingly big gulf between polynomial and exponential running times.

[9]In sorting, n denotes the length of the input array; in the four graph problems, n and m denote the number of vertices and edges, respectively; and in the sequence alignment problem, n and m denote the lengths of the two input strings.

[10]Remember that a logarithmic factor can be bounded above (sloppily) by a linear factor; for example, if $T(n) = O(n \log n)$, then $T(n) = O(n^2)$ as well.



Imagine that you have a fixed time budget, like an hour or a day. How does the solvable input size scale with additional computing power? With a polynomial-time algorithm, it increases by a constant factor (such as from 1,000,000 to 1,414,213) with every doubling of your computing power.[11] With an algorithm that runs in time proportional to 2^n, where n is the input size, each doubling of computing power increases the solvable input size by only one (such as from 1,000,000 to 1,000,001)!

19.3.3 Easy Problems

The theory of NP-hardness defines "easy" problems as those solvable by a polynomial-time algorithm, or equivalently by an algorithm for which the solvable input size (for a fixed time budget) scales multiplicatively with increasing computational power:[12]

Polynomial-Time Solvable Problems

A computational problem is *polynomial-time solvable* if there is a polynomial-time algorithm that solves it correctly for every input.

[11]With a linear-time algorithm, you could solve problems that are twice as big; with a quadratic-time algorithm, $\sqrt{2} \approx 1.414$ times as big; with a cubic-time algorithm, $\sqrt[3]{2} \approx 1.26$ as big; and so on.

[12]This definition was proposed independently by Alan Cobham and Jack Edmonds (see footnote 6) in the mid-1960s.

For example, the six problems listed at the beginning of this section are all polynomial-time solvable.

Technically, a (useless-in-practice) algorithm that runs in $O(n^{100})$ time on size-n inputs counts as a polynomial-time algorithm, and a problem solved by such an algorithm qualifies as polynomial-time solvable. Turning this statement around, if a problem like the TSP is *not* polynomial-time solvable, there is not even an $O(n^{100})$-time or $O(n^{10000})$-time algorithm that solves it (!).

Courage, Definitions, and Edge Cases

The identification of "easy" with "polynomial-time solvable" is imperfect; a problem might be solved in theory (by an algorithm that technically runs in polynomial time) but not in reality (by an empirically fast algorithm), or vice versa. Anyone with the guts to write down a precise mathematical definition (like polynomial-time solvability) to express a messy real-world concept (like "easy to solve via computer in the physical world") must be ready for friction between the binary nature of the definition and the fuzziness of reality. The definition will inevitably include or exclude some edge cases that you wish had gone the other way, but this is no excuse to ignore or dismiss a good definition. Polynomial-time solvability has been unreasonably effective at classifying problems as "easy" or "hard" in a way that accords with empirical experience. With a half-century of evidence behind us, we can confidently say that natural polynomial-time solvable problems typically can be solved with practical general-purpose algorithms, and that problems believed to not be polynomial-time solvable typically require significantly more work and domain expertise.

19.3.4 Relative Intractability

Suppose we suspected that a problem like the TSP is "not easy," meaning unsolvable by any polynomial-time algorithm (no matter how large the polynomial). How would we amass evidence that this is, in fact, the case? The most convincing argument, of course, would be an airtight mathematical proof. But the status of the TSP remains in limbo to this day: No one has found a polynomial-time algorithm that solves it, nor has anyone found a proof that no such algorithm exists.

How can we develop a theory that usefully differentiates "tractable" and "intractable" problems despite our deficient understanding of what algorithms can do? The brilliant conceit behind the theory of NP-hardness is to classify problems based on their *relative* (rather than absolute) difficulty and to declare a problem as "hard" if it is "at least as hard as" an overwhelming number of other unsolved problems.

19.3.5 Hard Problems

The many failed attempts at solving the TSP (Section 19.1.3) provide circumstantial evidence that the problem may not be polynomial-time solvable.

Weak Evidence of Hardness

A polynomial-time algorithm for the TSP would solve a problem that has resisted the efforts of hundreds (if not thousands) of brilliant minds over many decades.

Can we do better, meaning build a more compelling case of intractability? This is where the magic and power of NP-hardness comes in. The big idea is to show that a problem like the TSP is at least as hard as a vast array of unsolved problems from many different scientific fields—in fact, all problems for which you quickly know a solution when you see one. Such an argument would imply that a hypothetical polynomial-time algorithm for the TSP would automatically solve all these other unsolved problems, as well!

Strong Evidence of Hardness

A polynomial-time algorithm for the TSP would solve *thousands* of problems that have resisted the efforts of *tens (if not hundreds) of thousands* of brilliant minds over many decades.

In effect, the theory of NP-hardness shows that thousands of computational problems (including the TSP) are variations of the same problem in disguise, all destined to suffer identical computational fates. If you're trying to devise a polynomial-time algorithm for an NP-hard problem like the TSP, you're inadvertently attempting to also come up with such algorithms for these thousands of related problems.[13]

We call a problem *NP-hard* if there is strong evidence of intractability in the sense above:

NP-Hardness (Main Idea)

A problem is *NP-hard* if it is at least as difficult as every problem with easily recognized solutions.

This idea will be made 100% precise in Section 23.3.4; until then, we'll work with a provisional definition of NP-hardness that is phrased in terms of a famous mathematical conjecture, the "P $\neq$ NP conjecture."

19.3.6 The P $\neq$ NP Conjecture

Perhaps you've heard of the P $\neq$ NP conjecture. What is it, exactly? Section 23.4 provides the precise mathematical statement; for now, we'll settle for an informal version that should resonate with anyone who's had to grade student homework:

The P $\neq$ NP Conjecture (Informal Version)

Checking an alleged solution to a problem can be fundamentally easier than coming up with your own solution from scratch.

[13]Playing devil's advocate, hundreds (if not thousands) of brilliant minds have likewise failed to prove the other direction, that the TSP is *not* polynomial-time solvable. Symmetrically, doesn't this suggest that perhaps no such proof exists? The difference is that we seem far better at proving solvability (with fast algorithms known for countless problems) than unsolvability. Thus, if the TSP were polynomial-time solvable, it would be odd that we haven't yet found a polynomial-time algorithm for it; if not, no surprise that we haven't yet figured out how to prove it.

The "P" and "NP" in the conjecture refer to problems that can be solved from scratch in polynomial time and those whose solutions can be checked in polynomial time, respectively; for formal definitions, see Chapter 23.

For example, checking someone's proposed solution to a Sudoku or KenKen puzzle sure seems easier than working it out yourself. Or, in the context of the TSP, it's easy to verify that someone's proposed traveling salesman tour is good (with a total cost of, say, at most 1000) by adding up the costs of its edges; it's not so clear how you would quickly come up with your own such tour from scratch. Thus, intuition strongly suggests that the P $\neq$ NP conjecture is true.[14,15]

19.3.7 Provisional Definition of NP-Hardness

Provisionally, we'll call a problem NP-hard if, assuming that the P $\neq$ NP conjecture *is* true, it cannot be solved by any polynomial-time algorithm.

NP-Hard Problem (Provisional Definition)

A computational problem is *NP-hard* if a polynomial-time algorithm solving it would refute the P $\neq$ NP conjecture.

Thus, any polynomial-time algorithm for any NP-hard problem (such as the TSP) would automatically imply that the P $\neq$ NP conjecture is false and trigger an algorithmic bounty that seems too good to be true: a polynomial-time algorithm for every single problem for which solutions can be recognized in polynomial time. In the likely event that the P $\neq$ NP conjecture is true, no NP-hard problem is polynomial-time solvable, not even with an algorithm that runs in $O(n^{100})$ or $O(n^{10000})$ time on size-n inputs.

[14]We'll see in Problem 23.2 that the P $\neq$ NP conjecture is equivalent to Edmonds's conjecture (page 6) stating that the TSP cannot be solved in polynomial time.

[15]Why isn't it "obvious" that the P $\neq$ NP conjecture is true? Because the space of polynomial-time algorithms is unfathomably rich, with many ingenious inhabitants. (Perhaps you've come across Strassen's mind-blowing subcubic matrix multiplication algorithm, for example in Chapter 3 of *Part 1*?) Proving that none of the infinitely many candidate algorithms solve the TSP seems pretty intimidating!

19.3.8 Randomized and Quantum Algorithms

Our definition of polynomial-time solvability on page 11 contemplates only deterministic algorithms. As we know, randomization can be a powerful tool in algorithm design (for example, in the QuickSort algorithm). Can randomized algorithms escape the binds of NP-hardness?

More generally, what about much-hyped quantum algorithms? (As it turns out, randomized algorithms can be viewed as a special case of quantum algorithms.) It's true that large-scale, general-purpose quantum computers (if realized) would be a game-changer for a handful of problems, including the extremely important problem of factoring large integers. However, the factoring problem is not known or believed to be NP-hard, and experts conjecture that even quantum computers cannot solve NP-hard problems in polynomial time. The challenges posed by NP-hardness are not going away anytime soon.[16]

19.3.9 Subtleties

The oversimplified discussion at the beginning of this section (page 9) suggested that a "hard" problem would require exponential time to solve in the worst case. Our provisional definition in Section 19.3.7 says something different: An NP-hard problem is one that, assuming the P $\neq$ NP conjecture, cannot be solved by any polynomial-time algorithm.

The first discrepancy between the two definitions is that NP-hardness rules out polynomial-time solvability only if the P $\neq$ NP conjecture is true (and this remains an open question). If the conjecture is false, almost all the NP-hard problems discussed in this book are, in fact, polynomial-time solvable.

The second discrepancy is that, even in the likely event that the P $\neq$ NP conjecture is true, NP-hardness implies only that super-

[16]A majority of experts believe that every polynomial-time randomized algorithm can be *derandomized* and turned into an equivalent polynomial-time deterministic algorithm (perhaps with a larger polynomial in the running time bound). If true, the P $\neq$ NP conjecture would automatically apply to randomized algorithms as well.

By contrast, a majority of experts believe that quantum algorithms *are* fundamentally more powerful than classical algorithms (but not powerful enough to solve NP-hard problems in polynomial time). Isn't it amazing—and exciting—how much we still don't know?

polynomial (as opposed to exponential) time is required in the worst case to solve the problem.[17] However, for most natural NP-hard problems, including all those studied in this book, experts generally believe that exponential time is indeed required in the worst case. This belief is formalized by the "Exponential Time Hypothesis," a stronger form of the P $\neq$ NP conjecture (see Section 23.5).[18]

Finally, while 99% of the problems that you'll come across will be either "easy" (polynomial-time solvable) or "hard" (NP-hard), a few rare examples appear to lie in between. Thus, our "dichotomy" between easy and hard problems covers most, but not all, practically relevant computational problems.[19]

19.4 Algorithmic Strategies for NP-Hard Problems

Suppose you've identified a computational problem on which the success of your project rests. Perhaps you've spent the last several weeks throwing the kitchen sink at it—all the algorithm design paradigms you know, every data structure in the book, all the for-free primitives—but nothing works. Finally, you realize that the issue is not a deficiency of ingenuity on your part, it's the fact that the problem is NP-hard. Now you have an explanation of why your weeks of effort have come to naught, but that doesn't diminish the problem's significance to your project. What should you do?

19.4.1 General, Correct, Fast (Pick Two)

The bad news is that NP-hard problems are ubiquitous; right now, one might well be lurking in your latest project. The good news is that NP-hardness is not a death sentence. NP-hard problems can often

[17]Examples of running time bounds that are super-polynomial but subexponential in the input size n include $n^{\log_2 n}$ and $2^{\sqrt{n}}$.

[18]None of the computational problems studied in this book series require more than exponential time to solve, but other problems do. One famous example is the "halting problem," which can't be solved in any finite (let alone exponential) amount of time; see also Section 23.1.2.

[19]Two important problems that are believed to be neither polynomial-time solvable nor NP-hard are factoring (finding a non-trivial factor of an integer or determining that none exist) and the graph isomorphism problem (determining whether two graphs are identical up to a renaming of the vertices). Subexponential-time (but not polynomial-time) algorithms are known for both problems.

(but not always) be solved in practice, at least approximately, through sufficient investment of resources and algorithmic sophistication.

NP-hardness throws down the gauntlet to the algorithm designer and tells you where to set your expectations. You should not expect a general-purpose and always-fast algorithm for an NP-hard problem, akin to those we've seen for problems such as sorting, shortest paths, or sequence alignment. Unless you're lucky enough to face only unusually small or well-structured inputs, you're going to have to work pretty hard to solve the problem, and possibly also make some compromises.

What kinds of compromises? NP-hardness rules out algorithms with the following three desirable properties (assuming the $P \neq NP$ conjecture):

Three Properties (You Can't Have Them All)

1. *General-purpose.* The algorithm accommodates all possible inputs of the computational problem.
2. *Correct.* For every input, the algorithm correctly solves the problem.
3. *Fast.* For every input, the algorithm runs in polynomial time.

Accordingly, you can choose from among three types of compromises: compromising on generality, compromising on correctness, and compromising on speed. All three strategies are useful and common in practice.

The rest of this section elaborates on these three algorithmic strategies; Chapters 20 and 21 are deep dives into the latter two. As always, our focus is on powerful and flexible algorithm design principles that apply to a wide range of problems. You should take these principles as a starting point and run with them, guided by whatever domain expertise you have for the specific problem that you need to solve.

19.4.2 Compromising on Generality

One strategy for making progress on an NP-hard problem is to give up on general-purpose algorithms and focus instead on special cases of the problem relevant to your application. In the best-case scenario, you can identify domain-specific constraints on inputs and design an algorithm that is always correct and always fast on this subset of inputs. Graduates of the dynamic programming boot camp in *Part 3* have already seen two examples of this strategy.

Weighted independent set. In this problem, the input is an undirected graph $G = (V, E)$ and a nonnegative weight w_v for each vertex $v \in V$; the goal is to compute an independent set $S \subseteq V$ with the maximum-possible sum $\sum_{v \in S} w_v$ of vertex weights, where an *independent set* is a subset $S \subseteq V$ of mutually non-adjacent vertices (with $(v, w) \notin E$ for every $v, w \in S$). For example, if edges represent conflicts (between people, courses, etc.), independent sets correspond to conflict-free subsets. This problem is NP-hard in general, as we'll see in Section 22.5. The special case of the problem in which G is a path graph (with vertices $v_1, v_2, \ldots, v_n$ and edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$) can be solved in linear time using a dynamic programming algorithm. This algorithm can be extended to accommodate all acyclic graphs (see Problem 16.3 of *Part 3*).

Knapsack. In this problem, the input is specified by $2n + 1$ positive integers: n item values $v_1, v_2, \ldots, v_n$, n item sizes $s_1, s_2, \ldots, s_n$, and a knapsack capacity C. The goal is to compute a subset $S \subseteq \{1, 2, \ldots, n\}$ of items with the maximum-possible sum $\sum_{i \in S} v_i$ of values, subject to having total size $\sum_{i \in S} s_i$ at most C. In other words, the objective is to make use of a scarce resource in the most valuable way possible.[20] This problem is NP-hard, as we'll see in Section 22.8 and Problem 22.7. There is an $O(nC)$-time dynamic programming algorithm for the problem; this is a polynomial-time algorithm in the special case in which C is bounded by a polynomial function of n.

A Polynomial-Time Algorithm for Knapsack?

Why doesn't the $O(nC)$-time algorithm for the knapsack problem refute the P $\neq$ NP conjecture? Because this is not a polynomial-time algorithm. The input

[20]For example, on which goods and services should you spend your paycheck to get the most value? Or, given an operating budget and a set of job candidates with differing productivity levels and requested salaries, whom should you hire?

size—the number of keystrokes needed to specify the input to a computer—scales with the number of *digits* in a number, not the *magnitude* of a number. It doesn't take a million keystrokes to communicate the number "1,000,000"—only 7 (or 20 if you're working base-2). For example, in an instance with n items, knapsack capacity 2^n, and all item values and sizes at most 2^n, the input size is $O(n^2)$—$O(n)$ numbers with $O(n)$ digits each—while the running time of the dynamic programming algorithm is exponentially larger (proportional to $n \cdot 2^n$).

The algorithmic strategy of designing fast and correct algorithms (for special cases) uses the entire algorithmic toolbox that we developed in *Parts 1–3*. For this reason, no chapter of this book is dedicated to this strategy. We will, however, encounter along the way further examples of polynomial-time solvable special cases of NP-hard problems, including the traveling salesman, satisfiability, and graph coloring problems (see Problems 19.8 and 21.12).

19.4.3 Compromising on Correctness

The second algorithmic strategy, which is particularly popular in time-critical applications, is to insist on generality and speed at the expense of correctness. Algorithms that are not always correct are sometimes called *heuristic algorithms*.[21]

Ideally, a heuristic algorithm is "mostly correct." This could mean one or both of two things:

Relaxations of Correctness

1. The algorithm is correct on "most" inputs.[22]
2. The algorithm is "almost correct" on every input.

[21]In *Parts 1–3*, there is exactly one example of a mostly-but-not-always-correct solution: bloom filters, a small-space data structure that supports super-fast insertions and lookups, at the expense of occasional false positives.

[22]For example, one typical implementation of a bloom filter has a 2% false positive rate, with 98% of lookups answered correctly.

The second property is easiest to interpret for optimization problems, in which the goal is to compute a feasible solution (like a traveling salesman tour) with the best objective function value (like the minimum total cost). "Almost correct" then means that the algorithm outputs a feasible solution with objective function value close to the best possible, like a traveling salesman tour with total cost not much more than that of an optimal tour.

Your existing algorithmic toolbox for designing fast exact algorithms is directly useful for designing fast heuristic algorithms. For example, Sections 20.1–20.3 describe greedy heuristics for problems ranging from scheduling to influence maximization in social networks. These heuristic algorithms come with proofs of "approximate correctness" guaranteeing that, for every input, the objective function value of the algorithm's output is within a modest constant factor of the best-possible objective function value.[23]

Sections 20.4–20.5 augment your toolbox with the *local search* algorithm design paradigm. Local search and its generalizations are unreasonably effective in practice at tackling many NP-hard problems, including the TSP, even though local search algorithms rarely possess compelling approximate correctness guarantees.

19.4.4 Compromising on Worst-Case Running Time

The final strategy is appropriate for applications in which you cannot afford to compromise on correctness and are therefore unwilling to consider heuristic algorithms. Every correct algorithm for an NP-hard problem must run in super-polynomial time on some inputs (assuming the P $\neq$ NP conjecture). The goal, therefore, is to design an algorithm that is as fast as possible—at a minimum, one that improves dramatically on naive exhaustive search. This could mean one or both of two things:

Relaxations of Polynomial Running Time

1. The algorithm typically runs quickly (for example, in polynomial time) on the inputs that are relevant to

[23]Some authors call such algorithms "approximation algorithms" while reserving the term "heuristic algorithms" for algorithms that lack such proofs of approximate correctness.

your application.

2. The algorithm is faster than exhaustive search on every input.

In the second case, we should still expect the algorithm to run in exponential time on some inputs—after all, the problem is NP-hard. For example, Section 21.1 employs dynamic programming to beat exhaustive search for the TSP, reducing the running time from $O(n!)$ to $O(n^2 \cdot 2^n)$, where n is the number of vertices. Section 21.2 combines randomization with dynamic programming to beat exhaustive search for the problem of finding long paths in graphs (with running time $O((2e)^k \cdot m)$ rather than $O(n^k)$, where n and m denote the number of vertices and edges in the input graph, k the target path length, and $e = 2.718\ldots$).

Making progress on relatively large instances of NP-hard problems typically requires additional tools that do not possess better-than-exhaustive-search running time guarantees but are unreasonably effective in many applications. Sections 21.3–21.5 outline how to stand on the shoulders of experts who, over several decades, have developed remarkably potent solvers for mixed integer programming ("MIP") and satisfiability ("SAT") problems. Many NP-hard optimization problems (such as the TSP) can be encoded as mixed integer programming problems. Many NP-hard feasibility-checking problems (such as checking for a conflict-free assignment of classes to classrooms) are easily expressed as satisfiability problems. Whenever you face an NP-hard problem that can be easily specified as a MIP or SAT problem, try applying the latest and greatest solvers to it. There's no guarantee that a MIP or SAT solver will solve your particular instance in a reasonable amount of time—the problem is NP-hard, after all—but they constitute cutting-edge technology for tackling NP-hard problems in practice.

19.4.5 Key Takeaways

If you're shooting for level-1 knowledge of NP-hardness (Section 19.2), the most important things to remember are:

Three Facts About NP-Hard Problems

1. *Ubiquity:* Practically relevant NP-hard problems are everywhere.
2. *Intractability:* Under a widely believed mathematical conjecture, no NP-hard problem can be solved by any algorithm that is always correct and always runs in polynomial time.
3. *Not a death sentence:* NP-hard problems can often (but not always) be solved in practice, at least approximately, through sufficient investment of resources and algorithmic sophistication.

19.5 Proving NP-Hardness: A Simple Recipe

How can you recognize NP-hard problems when they arise in your own work, so that you can adjust your ambitions accordingly and abandon the search for an algorithm that is general-purpose, correct, and fast? Nobody wins if you spend weeks or months of your life inadvertently trying to refute the P $\neq$ NP conjecture.

First, know a collection of simple and common NP-hard problems (like the 19 problems in Chapter 22); in the simplest scenario, your application will literally boil down to one of these problems. Second, sharpen your ability to spot reductions between computational problems. Reducing one problem to another can spread computational tractability from the latter to the former. Turning this statement on its head, such a reduction can also spread computational *intractability* in the opposite direction, from the former problem to the latter. Thus, to show that a computational problem that you care about is NP-hard, all you need to do is reduce a known NP-hard problem to it.

The rest of this section elaborates on these points and provides one simple example; for a deep dive, see Chapter 22.

19.5.1 Reductions

Any problem B that is at least as hard as an NP-hard problem A is itself NP-hard. The phrase “at least as hard as” can be formalized