Algorithms

Jeff Erickson

0th edition (pre-publication draft) — December 30, 2018
½th edition (pre-publication draft) — April 9, 2019
1st paperback edition — June 13, 2019

1 2 3 4 5 6 7 8 9 — 27 26 25 24 23 22 21 20 **19**

ISBN: 978-1-792-64483-2 (paperback)

Download this book at http://jeffe.cs.illinois.edu/teaching/algorithms/
or http://algorithms.wtf
or https://archive.org/details/Algorithms-Jeff-Erickson

Please report errors at https://github.com/jeffgerickson/algorithms

Portions of our programming are mechanically reproduced,
and we now begin our broadcast day.

For Kim, Kay, and Hannah
with love and admiration

And for Erin
with thanks
for breaking her promise

Incipit prologus in libro alghoarismi de practica arismetrice.

— Ioannis Hispalensis [John of Seville?],
Liber algorismi de pratica arismetrice (c.1135)

Shall I tell you, my friend, how you will come to understand it?
Go and write a book upon it.

— Henry Home, Lord Kames (1696–1782),
in a letter to Sir Gilbert Elliot

The individual is always mistaken. He designed many things, and drew in other persons as coadjutors, quarrelled with some or all, blundered much, and something is done; all are a little advanced, but the individual is always mistaken. It turns out somewhat new and very unlike what he promised himself.

— Ralph Waldo Emerson, "Experience", *Essays, Second Series* (1844)

What I have outlined above is the content of a book the realization of whose basic plan and the incorporation of whose details would perhaps be impossible; what I have written is a second or third draft of a preliminary version of this book

— Michael Spivak, preface of the first edition of
Differential Geometry, Volume I (1970)

Preface

About This Book

This textbook grew out of a collection of lecture notes that I wrote for various algorithms classes at the University of Illinois at Urbana-Champaign, which I have been teaching about once a year since January 1999. Spurred by changes of our undergraduate theory curriculum, I undertook a major revision of my notes in 2016; this book consists of a subset of my revised notes on the most fundamental course material, mostly reflecting the algorithmic content of our new required junior-level theory course.

Prerequisites

The algorithms classes I teach at Illinois have two significant prerequisites: a course on discrete mathematics and a course on fundamental data structures. Consequently, this textbook is probably not suitable for most students as a *first*

course in data structures and algorithms. In particular, I assume at least passing familiarity with the following specific topics:

- **Discrete mathematics:** High-school algebra, logarithm identities, naive set theory, Boolean algebra, first-order predicate logic, sets, functions, equivalences, partial orders, modular arithmetic, recursive definitions, trees (as abstract objects, not data structures), graphs (vertices and edges, not function plots).
- **Proof techniques:** direct, indirect, contradiction, exhaustive case analysis, and induction (especially "strong" and "structural" induction). Chapter o uses induction, and whenever Chapter $n-1$ uses induction, so does Chapter n.
- **Iterative programming concepts:** variables, conditionals, loops, records, indirection (addresses/pointers/references), subroutines, recursion. I do not assume fluency in any particular programming language, but I do assume experience with at least one language that supports both indirection and recursion.
- **Fundamental abstract data types:** scalars, sequences, vectors, sets, stacks, queues, maps/dictionaries, ordered maps/dictionaries, priority queues.
- **Fundamental data structures:** arrays, linked lists (single and double, linear and circular), binary search trees, at least one form of *balanced* binary search tree (such as AVL trees, red-black trees, treaps, skip lists, or splay trees), hash tables, binary heaps, and most importantly, the difference between this list and the previous list.
- **Fundamental computational problems:** elementary arithmetic, sorting, searching, enumeration, tree traversal (preorder, inorder, postorder, level-order, and so on).
- **Fundamental algorithms:** elementary algorism, sequential search, binary search, sorting (selection, insertion, merge, heap, quick, radix, and so on), breadth- and depth-first search in (at least binary) trees, and most importantly, the difference between this list and the previous list.
- **Elementary algorithm analysis:** Asymptotic notation (o, O, Θ, Ω, ω), translating loops into sums and recursive calls into recurrences, evaluating simple sums and recurrences.
- **Mathematical maturity:** facility with abstraction, formal (especially recursive) definitions, and (especially inductive) proofs; writing and following mathematical arguments; recognizing and avoiding syntactic, semantic, and/or logical nonsense.

The book *briefly* covers some of this prerequisite material when it arises in context, but more as a reminder than a good introduction. For a more thorough overview, I strongly recommend the following freely available references:

- Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*. Version 1.3 (January 2013) or later available from http://mfleck.cs.illinois.edu/building-blocks/.
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. *Mathematics for Computer Science*. June 2018 revision available from https://courses.csail.mit.edu/6.042/spring18/. (I strongly recommend searching for the most recent revision.)
- Pat Morin. *Open Data Structures*. Edition 0.1Gβ (January 2016) or later available from http://opendatastructures.org/.
- Don Sheehy. *A Course in Data Structures and Object-Oriented Design*. February 2019 or later revision available from https://donsheehy.github.io/datastructures/.

Additional References

Please do not restrict yourself to this or any other single reference. Authors and readers bring their own perspectives to any intellectual material; no instructor "clicks" with every student, or even with every very strong student. Finding the author that most effectively gets *their* intuition into *your* head takes some effort, but that effort pays off handsomely in the long run.

The following references have been particularly valuable sources of intuition, examples, exercises, and inspiration; this is not meant to be a complete list.

- Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (I used this textbook as an undergraduate at Rice and again as a masters student at UC Irvine.)
- Boaz Barak. *Introduction to Theoretical Computer Science*. Textbook draft, most recently revised June 2019. (Not your grandfather's theoretical CS textbook, and so much the better for it; the fact that it's free is a delightful bonus.)
- Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein. *Introduction to Algorithms*, third edition. MIT Press/McGraw-Hill, 2009. (I used the first edition as a teaching assistant at Berkeley.)
- Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. (Probably the closest in content to this book, but considerably less verbose.)
- Jeff Edmonds. *How to Think about Algorithms*. Cambridge University Press, 2008.
- Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

- Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.
- Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005. Borrow it from the library if you can.
- Donald Knuth. *The Art of Computer Programming*, volumes 1–4A. Addison-Wesley, 1997 and 2011. (My parents gave me the first three volumes for Christmas when I was 14. Alas, I didn't actually read them until *much* later.)
- Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. (I used this textbook as a teaching assistant at Berkeley.)
- Ian Parberry. *Problems on Algorithms*. Prentice-Hall, 1995 (out of print). Downloadable from https://larc.unt.edu/ian/books/free/license.html after you agree to make a small charitable donation. Please honor your agreement.
- Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.
- Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- Class notes from my own algorithms classes at Berkeley, especially those taught by Dick Karp and Raimund Seidel.
- Lecture notes, slides, homeworks, exams, video lectures, research papers, blog posts, StackExchange questions and answers, podcasts, and full-fledged MOOCs made freely available on the web by innumerable colleagues around the world.

About the Exercises

Each chapter ends with several exercises, most of which I have used at least once in a homework assignment, discussion/lab section, or exam. The exercises are *not* ordered by increasing difficulty, but (generally) clustered by common techniques or themes. Some problems are annotated with symbols as follows:

- ♥Red hearts indicate particularly challenging problems; many of these have appeared on qualifying exams for PhD students at Illinois. A small number of *really* hard problems are marked with ♥large hearts.
- ♦Blue diamonds indicate problems that require familiarity with material from later chapters, but thematically belong where they are. Problems that require familiarity with *earlier* material are not marked, however; the book, like life, is cumulative.
- ♣Green clubs indicate problems that require familiarity with material outside the scope of this book, such as finite-state machines, linear algebra, probability, or planar graphs. These are rare.
- ♠Black spades indicate problems that require a significant amount of grunt work and/or coding. These are rare.

- ★Orange stars indicate that you are eating Lucky Charms that were manufactured before 1998. Ew.

These exercises are designed as opportunities to practice, not as targets for their own sake. The goal of each problem is not to solve that specific problem, but to exercise a certain set of skills, or to practice solving a certain *type* of problem. Partly for this reason, I don't provide solutions to the exercises; the solutions are not the point. In particular, there is no "instructor's manual"; if you can't solve a problem yourself, you probably shouldn't assign it to your students. That said, you can probably find solutions to whatever homework problems I've assigned *this* semester on the web page of whatever course I'm teaching. And nothing is stopping *you* from writing an instructor's manual!

Steal This Book!

 A complete electronic version of this book is freely available at any of the following locations:

- The book web site: http://jeffe.cs.illinois.edu/teaching/algorithms/
- The mnemonic shortcut: http://algorithms.wtf
- The bug-report site: https://github.com/jeffgerickson/algorithms
- The Internet Archive: https://archive.org/details/Algorithms-Jeff-Erickson

The book web site also contains several hundred pages of additional lecture notes on related and more advanced material, as well as a near-complete archive of past homeworks, exams, discussion/lab problems, and other teaching resources. Whenever I teach an algorithms class, I revise, update, and sometimes cull my teaching materials, so you may find more recent revisions on the web page of whatever course I am currently teaching.

Whether you are a student or an instructor, you are more than welcome to use any subset of this textbook or my other lecture notes in your own classes, without asking my permission—that's why I put them on the web! However, please also cite this book, either by name or with a link back to http://algorithms.wtf; this is *especially* important if you are a student, and you use my course materials to help with your homework. (Please also check with your instructor.)

However, if you are an instructor, I strongly encourage you to supplement these with additional material *that you write yourself*. Writing the material yourself will strengthen your mastery and in-class presentation of the material, which will in turn improve your students' mastery of the material. It will also get you past the frustration of dealing with the parts of this book that you don't like. All textbooks are ~~crap~~ imperfect, and this one is no exception.

Finally, **please make whatever you write freely, easily, and globally available on the open web**—not hidden behind the gates of a learning management system or some other type of paywall—so that students and instructors elsewhere can benefit from your unique insights. In particular, if you develop useful resources that directly complement this textbook, such as slides, videos, or solution manuals, please let me know so that I can add links to your resources from the book web site.

Acknowledgments

This textbook draws heavily on the contributions of countless algorithms students, teachers, and researchers. In particular, I am immensely grateful to more than three thousand Illinois students who have used my lecture notes as a primary reference, offered useful (if sometimes painful) criticism, and suffered through some truly awful early drafts. Thanks also to many colleagues and students around the world who have used these notes in their own classes and have sent helpful feedback and bug reports.

I am particularly grateful for the feedback and contributions (especially exercises) from my amazing teaching assistants:

> Aditya Ramani, Akash Gautam, Alex Steiger, Alina Ene, Amir Nayyeri, Asha Seetharam, Ashish Vulimiri, Ben Moseley, Brad Sturt, Brian Ensink, Chao Xu, Charlie Carlson, Chris Neihengen, Connor Clark, Dan Bullok, Dan Cranston, Daniel Khashabi, David Morrison, Ekta Manaktala, Erin Wolf Chambers, Gail Steitz, Gio Kao, Grant Czajkowski, Hsien-Chih Chang, Igor Gammer, Jacob Laurel, John Lee, Johnathon Fischer, Junqing Deng, Kent Quanrud, Kevin Milans, Kevin Small, Konstantinos Koiliaris, Kyle Fox, Kyle Jao, Lan Chen, Mark Idleman, Michael Bond, Mitch Harris, Naveen Arivazhagen, Nick Bachmair, Nick Hurlburt, Nirman Kumar, Nitish Korula, Patrick Lin, Phillip Shih, Rachit Agarwal, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Sahand Mozaffari, Shalan Naqvi, Shripad Thite, Spencer Gordon, Srihita Vatsavaya, Subhro Roy, Tana Wattanawaroon, Umang Mathur, Vipul Goyal, Yasu Furakawa, and Yipu Wang.

I've also been helped tremendously by many discussions with faculty colleagues at Illinois: Alexandra Kolla, Cinda Heeren, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Kim Whittlesey, Lenny Pitt, Madhu Parasarathy, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, and especially Chandra Chekuri, Ed Reingold, and Sariel Har-Peled.

Of course this book owes a great debt to the people who taught me this algorithms stuff in the first place: Bob Bixby and Michael Pearlman at Rice; David Eppstein, Dan Hirschberg, and George Lueker at Irvine; and Abhiram Ranade, Dick Karp, Manuel Blum, Mike Luby, and Raimund Seidel at Berkeley.

I stole the first iteration of the overall course structure, and the idea to write up my own lecture notes in the first place, from Herbert Edelsbrunner; the idea of turning a subset of my notes into a book from Steve LaValle; and several components of the book design from Robert Ghrist.

Caveat Lector!

Of course, none of those people should be blamed for any flaws in the resulting book. Despite many rounds of revision and editing, this book contains several mistakes, bugs, gaffes, omissions, snafus, kludges, typos, mathos, grammaros, thinkos, brain farts, poor design decisions, historical inaccuracies, anachronisms, inconsistencies, exaggerations, dithering, blather, distortions, oversimplifications, redundancy, logorrhea, nonsense, garbage, cruft, junk, and outright lies, **all of which are entirely Steve Skiena's fault.**

I maintain an issue tracker at https://github.com/jeffgerickson/algorithms, where readers like you can submit bug reports, feature requests, and general feedback on the book. Please let me know if you find an error of any kind, whether mathematical, grammatical, historical, typographical, cultural, or otherwise, whether in the main text, in the exercises, or in my other course materials. (Steve is unlikely to care.) Of course, all other feedback is also welcome!

Enjoy!

— Jeff

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

— Steven S. Skiena, *The Algorithm Design Manual* (1997)

No doubt this statement will be followed by an annotated list of all textbooks, and why each one is crap.

— Adam Contini, MetaFilter, January 4, 2010

Table of Contents

Hinc incipit algorismus. *Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.*

— Friar Alexander de Villa Dei, *Carmen de Algorismo* (c. 1220)

You are right to demand that an artist engage his work consciously,
but you confuse two different things:
solving the problem and correctly posing the question.

— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The more we reduce ourselves to machines in the lower things,
the more force we shall set free to use in the higher.

— Anna C. Brackett, *The Technique of Rest* (1892)

And here I am at 2:30 a.m. writing about technique, in spite of a strong conviction that the moment a man begins to talk about technique that's proof that he is fresh out of ideas.

— Raymond Chandler, letter to Erle Stanley Gardner (May 5, 1939)

Good men don't need rules.
Today is not the day to find out why I have so many,

— The Doctor [Matt Smith], "A Good Man Goes to War", *Doctor Who* (2011)

0

Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions, usually intended to accomplish a specific purpose. For example, here is an algorithm for singing that annoying song "99 Bottles of Beer on the Wall", for arbitrary values of 99:

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

The word "algorithm" does *not* derive, as algorithmophobic classicists might guess, from the Greek roots *arithmos* (ἀριθμός), meaning "number", and *algos* (ἄλγος), meaning "pain". Rather, it is a corruption of the name of the 9th century Persian scholar Muḥammad ibn Mūsā al-Khwārizmī.[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*,[2] from which the modern word *algebra* derives. In a different treatise, al-Khwārizmī described the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *ṣifr* to represent a missing quantity—which had been developed in India several centuries earlier. The methods described in this latter treatise, using either written figures or counting stones, became known in English as *algorism* or *augrym*, and its figures became known in English as *ciphers*.

Although both place-value notation and al-Khwārizmī's works were already known by some European scholars, the "Hindu-Arabic" numeric system was popularized in Europe by the medieval Italian mathematician and tradesman Leonardo of Pisa, better known as Fibonacci. Thanks in part to his 1202 book *Liber Abaci*,[3] written figures began to replace the counting table (then known as an *abacus*) and finger arithmetic[4] as the preferred platform for calculation[5] in Europe in the 13th century—*not* because written decimal figures were easier to learn or use, but because they provided an audit trail. Ciphers became common in Western Europe only with the advent of movable type, and truly ubiquitous only after cheap paper became plentiful in the early 19th century.

Eventually the word *algorism* evolved into the modern *algorithm*, via folk etymology from the Greek *arithmos* (and perhaps the previously mentioned *algos*).[6] Thus, until very recently, the word *algorithm* referred exclusively

[1] "Mohammad, father of Adbdulla, son of Moses, the Kwārizmian". Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

[2] "The Compendious Book on Calculation by Completion and Balancing"

[3] While it is tempting to translate the title *Liber Abaci* as "The Book of the Abacus", a more accurate translation is "The Book of Calculation". Both before and after Fibonacci, the Italian word *abaco* was used to describe anything related to numerical calculation—devices, methods, schools, books, and so on—much in the same way that "computer science" is used today in English, or as the Chinese phrase for "operations research" translates literally as "the study of using counting rods".

[4] ☞ Reckoning with *digits*! ☜

[5] The word *calculate* derives from the Latin word *calculus*, meaning "small rock", referring to the stones on a counting table, or as Chaucer called them, *augrym stones*. In 440BCE, Herodotus wrote in his *Histories* that "The Greeks write and calculate (λογίζεσθαι ψήφοις, literally 'reckon with pebbles') from left to right; the Egyptians do the opposite. Yet they say that their way of writing is toward the right, and the Greek way toward the left." (Herodotus is strangely silent on which end of the egg the Egyptians ate first.)

[6] Some medieval sources claim that the Greek prefix "algo-" means "art" or "introduction". Others claim that algorithms were invented by a Greek philosopher, or a king of India, or perhaps a king of Spain, named "Algus" or "Algor" or "Argus". A few, possibly including Dante Alighieri, even identified the inventor with the mythological Greek shipbuilder and eponymous argonaut. It's unclear whether any of these risible claims were intended to be historically accurate, or merely mnemonic.

to mechanical techniques for place-value arithmetic using "Arabic" numerals. People trained in the fast and reliable execution of these procedures were called *algorists* or *computators*, or more simply, *computers*.

0.2 Multiplication

Although they have been a topic of formal academic study for only a few decades, algorithms have been with us since the dawn of civilization. Descriptions of step-by-step arithmetic computation are among the earliest examples of written human language, long predating the expositions by Fibonacci and al-Khwārizmī, or even the place-value notation they popularized.

Lattice Multiplication

The most familiar method for multiplying large numbers, at least for American students, is the ***lattice algorithm***. This algorithm was popularized by Fibonacci in *Liber Abaci*, who learned it from Arabic sources including al-Khwārizmī, who in turn learned it from Indian sources including Brahmagupta's 7th-century treatise *Brāhmasphuṭasiddhānta*, who may have learned it from Chinese sources. The oldest surviving descriptions of the algorithm appear in *The Mathematical Classic of Sunzi*, written in China between the 3rd and 5th centuries, and in Eutocius of Ascalon's commentaries on Archimedes' *Measurement of the Circle*, written around 500CE, but there is evidence that the algorithm was known much earlier. Eutocius credits the method to a lost treatise of Apollonius of Perga, who lived around 300BCE, entitled *Okytokion* (Ὠκυτόκιον).[7] The Sumerians recorded multiplication tables on clay tablets as early as 2600BCE, suggesting that they may have used the lattice algorithm.[8]

The lattice algorithm assumes that the input numbers are represented as explicit strings of digits; I'll assume here that we're working in base ten, but the algorithm generalizes immediately to any other base. To simplify notation,[9] the

[7]Literally "medicine that promotes quick and easy childbirth"! Pappus of Alexandria reproduced several excerpts of *Okytokion* about 200 years before Eutocius, but his description of the lattice multiplication algorithm (if he gave one) is *also* lost.

[8]There is ample evidence that ancient Sumerians calculated accurately with extremely large numbers using their base-60 place-value numerical system, but I am not aware of any surviving record of the actual methods they used. In addition to standard multiplication and reciprocal tables, tables listing the squares of integers from 1 to 59 have been found, leading some math historians to conjecture that Babylonians multiplied using an identity like $xy = ((x+y)^2 - x^2 - y^2)/2$. But this trick only works when $x + y < 60$; history is silent on how the Babylonians might have computed x^2 when $x \geq 60$.

[9]but at the risk of inflaming the historical enmity between Greece and Egypt, or Lilliput and Blefuscu, or Macs and PCs, or people who think zero is a natural number and people who are wrong

input consists of a pair of arrays $X[0..m-1]$ and $Y[0..n-1]$, representing the numbers

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{and} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j,$$

and similarly, the output consists of a single array $Z[0..m+n-1]$, representing the product

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

The algorithm uses addition and *single-digit* multiplication as primitive operations. Addition can be performed using a simple for-loop. In practice, single-digit multiplication is performed using a lookup table, either carved into clay tablets, painted on strips of wood or bamboo, written on paper, stored in read-only memory, or memorized by the computator. The entire lattice algorithm can be summarized by the formula

$$x \cdot y = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \left(X[i] \cdot Y[j] \cdot 10^{i+j} \right).$$

Different variants of the lattice algorithm evaluate the partial products $X[i] \cdot Y[j] \cdot 10^{i+j}$ in different orders and use different strategies for computing their sum. For example, in *Liber Abaco*, Fibonacci describes a variant that considers the mn partial products in increasing order of significance, as shown in modern pseudocode below.

```
FibonacciMultiply(X[0..m-1], Y[0..n-1]):
  hold ← 0
  for k ← 0 to n + m − 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n − 1]
```

Fibonacci's algorithm is often executed by storing all the partial products in a two-dimensional table (often called a "tableau" or "grate" or "lattice") and then summing along the diagonals with appropriate carries, as shown on the right in Figure 0.1. American elementary-school students are taught to multiply one factor (the "multiplicand") by each digit in the other factor (the "multiplier"), writing down all the multiplicand-by-digit products before adding them up, as shown on the left in Figure 0.1. This was also the method described by Eutocius, although he fittingly considered the multiplier digits from left to right, as shown

in Figure 0.2. Both of these variants (and several others) are described and illustrated side by side in the anonymous 1458 textbook *L'Arte dell'Abbaco*, also known as the *Treviso Arithmetic*, the first *printed* mathematics book in the West.

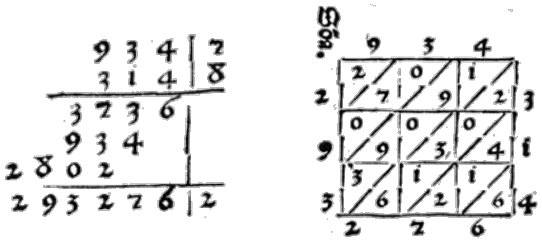


Figure 0.1. Computing $934 \times 314 = 293276$ using "long" multiplication (with error-checking by casting out nines) and "lattice" multiplication, from *L'Arte dell'Abbaco* (1458). (See Image Credits at the end of the book.)

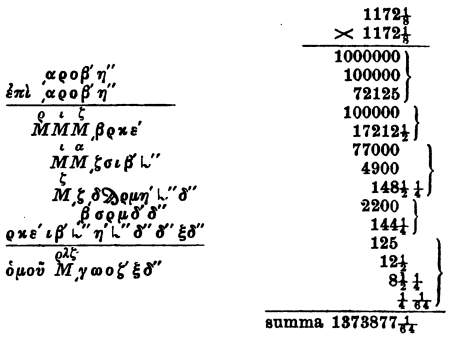


Figure 0.2. Eutocius's 6th-century calculation of $1172\frac{1}{8} \times 1172\frac{1}{8} = 1373877\frac{1}{64}$, in his commentary on Archimedes' *Measurement of the Circle*, transcribed (left) and translated into modern notation (right) by Johan Heiberg (1891). (See Image Credits at the end of the book.)

All of these variants of the lattice algorithm—and other similar variants described by Sunzi, al-Khwārizmī, Fibonacci, *L'Arte dell'Abbaco*, and many other sources—compute the product of any m-digit number and any n-digit number in $O(mn)$ ***time***; the running time of every variant is dominated by the number of single-digit multiplications.

Duplation and Mediation

The lattice algorithm is not the oldest multiplication algorithm for which we have direct recorded evidence. An even older and arguably simpler algorithm, which does not rely on place-value notation, is sometimes called *Russian peasant multiplication*, *Ethiopian peasant multiplication*, or just ***peasant multiplication***.A

variant of this algorithm was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650BCE, from a document he claimed was (then) about 350 years old.[10] This algorithm was still taught in elementary schools in Eastern Europe in the late 20th century; it was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

The peasant multiplication algorithm reduces the difficult task of multiplying arbitrary numbers to a sequence of four simpler operations: (1) determining parity (even or odd), (2) addition, (3) ***duplation*** (doubling a number), and (4) ***mediation*** (halving a number, rounding down).

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return prod
```

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

Figure 0.3. Multiplication by duplation and mediation

The correctness of this algorithm follows by induction from the following recursive identity, which holds for all non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{if } x \text{ is odd} \end{cases}$$

Arguably, this recurrence *is* the peasant multiplication algorithm. Don't let the iterative pseudocode fool you; the algorithm is fundamentally recursive!

As stated, PeasantMultiply performs $O(\log x)$ parity, addition, and mediation operations, but we can improve this bound to $O(\log \min\{x, y\})$ by swapping the two arguments when $x > y$. Assuming the numbers are represented using any reasonable place-value notation (like binary, decimal, Babylonian hexagesimal, Egyptian duodecimal, Roman numeral, Chinese counting rods, bead positions on an abacus, and so on), each operation requires at most $O(\log(xy)) = O(\log \max\{x, y\})$ single-digit operations, so the overall running time of the algorithm is $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

[10] The version of this algorithm actually used in ancient Egypt does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

In other words, this algorithm requires ***O(mn) time*** to multiply an m-digit number by an n-digit number; up to constant factors, this is the same running time as the lattice algorithm. This algorithm requires (a constant factor!) more paperwork to execute by hand than the lattice algorithm, but the necessary primitive operations are arguably easier for humans to perform. In fact, the two algorithms are equivalent when numbers are represented in binary.

Compass and Straightedge

Classical Greek geometers identified numbers (or more accurately, *magnitudes*) with line segments of the appropriate length, which they manipulated using two simple mechanical tools—the compass and the straightedge—versions of which had already been in common use by surveyors, architects, and other artisans for centuries. Using *only* these two tools, these scholars reduced several complex geometric constructions to the following primitive operations, starting with one or more identified reference points.

- Draw the unique line passing through two distinct identified points.
- Draw the unique circle centered at one identified point and passing through another.
- Identify the intersection point (if any) of two lines.
- Identify the intersection points (if any) of a line and a circle.
- Identify the intersection points (if any) of two circles.

In practice, Greek geometry students almost certainly drew their constructions on an *abax* (ἄβαξ), a table covered in dust or sand.[11] Centuries earlier, Egyptian surveyors carried out many of the same constructions using ropes to determine straight lines and circles on the ground.[12] However, Euclid and other Greek geometers presented compass and straightedge constructions as precise mathematical *abstractions*—points are *ideal* points; lines are *ideal* lines; and circles are *ideal* circles.

Figure 0.4 shows an algorithm, described in Euclid's *Elements* about 2500 years ago, for multiplying or dividing two magnitudes. The input consists of four distinct points A, B, C, and D, and the goal is to construct a point Z such that $|AZ| = |AC||AD|/|AB|$. In particular, if we define $|AB|$ to be our unit of length, then the algorithm computes the product of $|AC|$ and $|AD|$.

Notice that Euclid first defines a new primitive operation RIGHTANGLE by (as modern programmers would phrase it) writing a subroutine. The correctness

[11]The written numerals 1 through 9 were known in Europe at least two centuries before Fibonacci's *Liber Abaci* as "gobar numerals", from the Arabic word *ghubār* meaning dust, ultimately referring to the Indian practice of performing arithmetic on tables covered with sand. The Greek word ἄβαξ is the origin of the Latin *abacus*, which also originally referred to a sand table.

[12]Remember what "geometry" means? Democritus would later refer to these Egyptian surveyors, somewhat derisively, as *arpedonaptai* (ἀρπεδονάπται), meaning "rope-fasteners".


```
⟨⟨Construct the line perpendicular to ℓ passing through P.⟩⟩
RIGHTANGLE(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← INTERSECT(CIRCLE(P, A), ℓ)
  C, D ← INTERSECT(CIRCLE(A, B), CIRCLE(B, A))
  return LINE(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MULTIPLYORDIVIDE(A, B, C, D):
  α ← RIGHTANGLE(LINE(A, C), A)
  E ← INTERSECT(CIRCLE(A, B), α)
  F ← INTERSECT(CIRCLE(A, D), α)
  β ← RIGHTANGLE(LINE(E, C), F)
  γ ← RIGHTANGLE(β, F)
  return INTERSECT(γ, LINE(A, C))
```

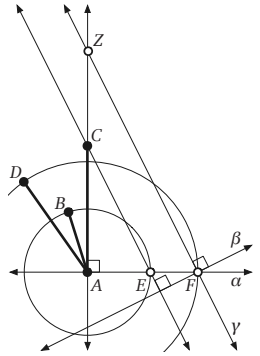


Figure 0.4. Multiplication by compass and straightedge.

of the algorithm follows from the observation that triangles ACE and AZF are similar. The second and third lines of the main algorithm are ambiguous, because α intersects any circle centered at A at *two* distinct points, but the algorithm is actually correct no matter which intersection points are chosen for E and F.

Euclid's algorithm reduces the problem of multiplying two magnitudes (lengths) to a series of primitive compass-and-straightedge operations. These operations are difficult to implement precisely on a modern digital computer, but Euclid's algorithm wasn't *designed* for a digital computer. It was designed for the Platonic Ideal Geometer, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge, who could execute each operation perfectly in constant time *by definition*. In this model of computation, MULTIPLYORDIVIDE runs in $O(1)$ time!

0.3 Congressional Apportionment

Here is another real-world example of an algorithm of significant political importance. Article I, Section 2 of the United States Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers.... The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative....

Because there are only a finite number of seats in the House of Representatives, *exact* proportional representation requires either shared or fractional representatives, neither of which are legal. As a result, over the next several decades, many different apportionment algorithms were proposed and used to round the ideal fractional solution fairly. The algorithm actually used today, called

the ***Huntington-Hill method*** or the ***method of equal proportions***, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §2a) in 1941, and survived a Supreme Court challenge in 1992.[13]

The Huntington-Hill method allocates representatives to states one at a time. First, in a preprocessing stage, each state is allocated one representative. Then in each iteration of the main loop, the next representative is assigned to the state with the highest *priority*. The priority of each state is defined to be $P/\sqrt{r(r+1)}$, where P is the state's population and r is the number of representatives already allocated to that state.

The algorithm is described in pseudocode in Figure 0.5. The input consists of an array $Pop[1..n]$ storing the populations of the n states and an integer R equal to the total number of representatives; the algorithm assumes $R \geq n$. (Currently, in the United States, $n = 50$ and $R = 435$.) The output array $Rep[1..n]$ records the number of representatives allocated to each state.

```
ApportionCongress(Pop[1..n], R):
  PQ ← NewPriorityQueue

  ⟨⟨Give every state its first representative⟩⟩
  for s ← 1 to n
      Rep[s] ← 1
      Insert(PQ, s, Pop[i]/√2)

  ⟨⟨Allocate the remaining n − R representatives⟩⟩
  for i ← 1 to n − R
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      priority ← Pop[s]/√(Rep[s](Rep[s] + 1))
      Insert(PQ, s, priority)

  return Rep[1..n]
```

Figure 0.5. The Huntington-Hill apportionment algorithm

This implementation of Huntington-Hill uses a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't say anything about priority queues, of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how this priority queue is implemented. The Census Bureau uses a sorted array, stored in a single column of an Excel spreadsheet, which is recalculated from scratch at every iteration. You (should have) learned a more efficient implementation in your undergraduate data structures class.

Similar apportionment algorithms are used in multi-party parliamentary elections around the world, where the number of seats allocated to each party is supposed to be proportional to the number of votes that party receives. The two most common are the *D'Hondt method*[14] and the *Webster–Sainte-Laguë method*,[15] which respectively use priorities $P/(r+1)$ and $P/(2r+1)$ in place of the square-root expression in Huntington-Hill. The Huntington-Hill method is essentially unique to the United States House of Representatives, thanks in part to the constitutional requirement that each state must be allocated at least one representative.

[13] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that ***any*** apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The current congressional apportionment algorithm is described in gruesome detail at the U.S. Census Department web site http://www.census.gov/topics/public-sector/congressional-apportionment.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.fas.org/sgp/crs/misc/R41382.pdf.

0.4 A Bad Example

As a prototypical example of a sequence of instructions that is *not* actually an algorithm, consider "Martin's algorithm":[16]

```
BeAMillionaireAndNeverPayTaxes():
  Get a million dollars.
  If the tax man comes to your door and says, "You have never paid taxes!"
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy! A group of billionaire CEOs, Silicon Valley venture capitalists, or New York City real-estate hustlers might consider this an algorithm, because for them the first step is both unambiguous and trivial,[17] but for the rest of us poor slobs, Martin's procedure is too vague to be considered an actual algorithm. On the other hand, this is a perfect example of a ***reduction***—it *reduces* the problem of being a millionaire and never paying taxes to the "easier" problem of acquiring a million dollars. We'll see reductions over and over again in this book. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

[14] developed by Thomas Jefferson in 1792, used for U.S. Congressional apportionment from 1792 to 1832, rediscovered by Belgian mathematician Victor D'Hondt in 1878, and refined by Swiss physicist Eduard Hagenbach-Bischoff in 1888.

[15] developed by Daniel Webster in 1832, used for U.S. Congressional apportionment from 1842 to 1911, rediscovered by French mathematician André Sainte-Laguë in 1910, and rediscovered again by German physicist Hans Schepers in 1980.

[16] Steve Martin, "You Can Be A Millionaire", *Saturday Night Live*, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

[17] Something something secure quantum blockchain deep-learning something.

Martin's algorithm, like some of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are difficult for computers to perform. This book focuses (almost!) exclusively on algorithms that can be reasonably implemented on a standard digital computer. Each step in these algorithms is either directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or something that you've already learned how to do (like sorting, binary search, tree traversal, or singing "*n* Bottles of Beer on the Wall").

0.5 Describing Algorithms

The skills required to effectively *design and analyze* algorithms are entangled with the skills required to effectively *describe* algorithms. At least in my classes, a complete description of any algorithm has four components:

- **What:** A precise specification of the problem that the algorithm solves.
- **How:** A precise description of the algorithm itself.
- **Why:** A proof that the algorithm solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to *develop* these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others. For example, we may need to tweak the problem description to support a faster algorithm, or modify the algorithm to handle a tricky case in the proof of correctness. Nevertheless, *presenting* these components separately is usually clearest for the reader.

As with any writing, it's important to aim your descriptions at the right audience; I recommend writing for a competent but skeptical programmer *who is not as clever as you are*. Think of yourself six months ago. As you develop any new algorithm, you will naturally build up lots of intuition about the problem and about how your algorithm solves it, and your informal reasoning will be guided by that intuition. But anyone *reading* your algorithm later, or the code you derive from it, won't share your intuition or experience. Neither will your compiler. Neither will you six months from now. All they will have is your written description.

Even if you never have to explain your algorithms to anyone else, it's still important to develop them with an audience in mind. Trying to communicate clearly forces you to *think* more clearly. In particular, writing for a *novice* audience, who will interpret your words *exactly* as written, forces you to work

through fine details, no matter how "obvious" or "intuitive" your high-level ideas may seem at the moment. Similarly, writing for a *skeptical* audience forces you to develop robust arguments for correctness and efficiency, instead of trusting your intuition or your intelligence.[18]

I cannot emphasize this point enough: **Your primary job as an algorithm designer is *teaching other people* how and why your algorithms work.** If you can't communicate your ideas to other human beings, they may as well not exist. Producing correct and efficient executable code is an important but secondary goal. Convincing yourself, your professors, your (prospective) employers, your colleagues, or your students that you are smart is at best a distant third.

Specifying the Problem

Before we can even start developing a new algorithm, we have to agree on what problem our algorithm is supposed to solve. Similarly, before we can even start *describing* an algorithm, we have to *describe* the problem that the algorithm is supposed to solve.

Algorithmic problems are often presented using standard English, in terms of real-world objects. It's up to us, the algorithm designers, to restate these problems in terms of formal, abstract, *mathematical* objects—numbers, arrays, lists, graphs, trees, and so on—that we can reason about formally. We must also determine if the problem statement carries any hidden assumptions, and state those assumptions explicitly. (For example, in the song "*n* Bottles of Beer on the Wall", *n* is always a non-negative integer.[19])

We may need to refine our specification as we develop the algorithm. For example, our algorithm may require a particular input representation, or produce a particular output representation, that was left unspecified in the original informal problem description. Or our algorithm might actually solve a *more general* problem than we were originally asked to solve. (This is a common feature of recursive algorithms.)

The specification should include just enough detail that someone else could *use* our algorithm as a black box, without knowing how or why the algorithm actually works. In particular, we must describe the type *and meaning* of each input parameter, and exactly how the eventual output depends on the input parameters. On the other hand, our specification should *deliberately hide* any details that are *not* necessary to use the algorithm as a black box. Let that which does not matter truly slide.

[18] In particular, I assume that *you* are a skeptical novice!

[19] I've never heard anyone sing "$\sqrt{2}$ Bottles of Beer on the Wall." Occasionally I *have* heard set theorists singing "$\aleph_0$ bottles of beer on the wall", but for some reason they always gave up before the song was over.

For example, the lattice and duplation-and-mediation algorithms both solve the same problem: Given two non-negative integers x and y, each represented as an array of digits, compute the product $x \cdot y$, also represented as an array of digits. To someone *using* these algorithms, the choice of algorithm is completely irrelevant. On the other hand, the Greek straightedge-and-compass algorithm solves a *different problem*, because the input and output values are represented by line segments instead of arrays of digits.

Describing the Algorithm

Computer programs are concrete representations of algorithms, but algorithms are *not* programs. Rather, algorithms are abstract mechanical procedures that can be implemented in *any* programming language that supports the underlying primitive operations. The idiosyncratic syntactic details of your favorite programming language are utterly irrelevant; focusing on these will only distract you (and your readers) from what's *really* going on.[20] A good algorithm description is closer to what we should write in the *comments* of a real program than the code itself. Code is a poor medium for storytelling.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have lots of idiomatic structure—especially conditionals, loops, function calls, and recursion—that are far too easily hidden by unstructured prose. Colloquial English is full of ambiguities and shades of meaning, but algorithms must be described as unambiguously as possible. Prose is a poor medium for precision.

In my opinion, the clearest way to present an algorithm is using a combination of *pseudo*code and structured English. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two, *whatever is clearest*. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm easier to understand, analyze, debug, and implement.

[20] This is, of course, a matter of religious conviction. Armchair linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate "jump the shark" or "Fortnite streamer" into Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are too subtle to have any impact on the material in *this* book. For a compelling counterexample, see Chris Okasaki's monograph *Functional Data Structures* and its more recent descendants.

Whenever we describe an algorithm, our description should include every detail necessary to fully specify the algorithm, prove its correctness, and analyze its running time. At the same time, it should *exclude* any details that are *not* necessary to fully specify the algorithm, prove its correctness, and analyze its running time. (Slide.) At a more practical level, our description should allow a competent but skeptical programmer *who has not read this book* to quickly and correctly implement the algorithm in *their* favorite programming language, *without understanding why it works*.

I don't want to bore you with the rules I follow for writing pseudocode, but I must caution against one especially pernicious habit. ***Never*** describe repeated operations informally, as in "Do [this] first, then do [that] second, ***and so on***." or "Repeat ***this process*** until [something]". As anyone who has taken one of those frustrating "What comes next in this sequence?" tests already knows, describing the first few steps of an algorithm says little or nothing about what happens in later steps. If your algorithm has a loop, write it as a loop, and explicitly describe what happens in an *arbitrary* iteration. Similarly, if your algorithm is recursive, write it recursively, and explicitly describe the case boundaries and what happens in each case.

0.6 Analyzing Algorithms

It's not enough just to write down an algorithm and say "Behold!" We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

In some application settings, it is acceptable for programs to behave correctly most of the time, on all "reasonable" inputs. Not in this book; we require algorithms that are *always* correct, for *all possible* inputs. Moreover, we must *prove* that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. Sometimes correctness is truly obvious, especially for algorithms you've seen in earlier courses. On the other hand, "obvious" is all too often a synonym for "wrong". Most of the algorithms we discuss in this course require real work to prove correct. In particular, correctness proofs usually involve induction. We *like* induction. Induction is our *friend*.[21]

Of course, before we can formally prove that our algorithm does what it's supposed to do, we have to formally describe what it's supposed to do!

[21] If induction is *not* your friend, you will have a hard time with this book.

Running Time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all "reasonable" inputs. Not in this book; we require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BottlesOfBeer(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Downloading an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BottlesOfBeer($2n$) requires about twice much time as singing BottlesOfBeer(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$.

We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the "code". For example, we might notice that the word "beer" is sung three times in every verse of BottlesOfBeer, so the number of times you sing "beer" is a good indication of the total singing time. For this question, we can give an exact answer: BottlesOfBeer(n) mentions beer exactly $3n+3$ times.

Incidentally, there are *lots* of songs with quadratic singing time. This one is probably familiar to most English-speakers:

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j],"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDaysOfChristmas is a list of $n-1$ gifts, represented here as an array. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts.

Other quadratic-time songs include "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus","The Barley-Mow", "If I Were Not Upon the Stage", "Star Trekkin'","Ist das nicht ein Schnitzelbank?",[22]"Il Pulcino Pio", "Minkurinn í hænsnakofanum", "Echad Mi Yodea", and "Το κοκοράκι". For more examples, consult your favorite preschooler.

```
Alouette(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerai. »
  pour tout i de 1 à n
      Chantez « Je te plumerai lapart[i]. Je te plumerai lapart[i]. »
      pour tout j de i à 1 ⟨⟨à rebours⟩⟩
          Chantez « Et lapart[j]! Et lapart[j]! »
      Chantez « Alouette! Alouette! Aaaaaa… »
      Chantez « …alouette, gentille allouette, alouette, je te plumerai. »
```

A few songs have even more bizarre singing times. A fairly modern example is "The TELNET Song" by Guy Steele, which actually takes $\Theta(2^n)$ time to sing the first n verses; Steele recommended $n = 4$. Finally, there are some songs that *never* end.[23]

Except for "The TELNET Song", all of these songs are most naturally expressed as a small set of nested loops, so their ~~running~~ singing times can be computed using nested summations. The running time of a *recursive* algorithm is more easily expressed as a recurrence. For example, the peasant multiplication algorithm can be expressed recursively as follows:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{if } x \text{ is odd} \end{cases}$$

Let $T(x,y)$ denote the number of parity, addition, and mediation operations required to compute $x \cdot y$. This function satisfies the recursive inequality $T(x,y) \le T(\lfloor x/2 \rfloor, 2y) + 2$ with base case $T(0,y) = 0$. Techniques described in the next chapter imply the upper bound $T(x,y) = O(\log x)$.

Sometimes the running time of an algorithm depends on a particular implementation of some underlying data structure of subroutine. For example, the Huntington-Hill apportionment algorithm ApportionCongress runs in $O(N + RI + (R-n)E)$ time, where N denotes the running time of NewPriorityQueue, I denotes the running time of Insert, and E denotes the running time

[22] Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!

[23] They just go on and on, my friend.

of ExtractMax. Under the reasonable assumption that $R \geq 2n$ (on average, each state gets at least two representatives), we can simplify this bound to $O(N + R(I + E))$. The precise running time depends on the implementation of the underlying priority queue. The Census Bureau implements the priority queue as an unsorted array, which gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the Census Bureau's implementation of ApportionCongress runs in $\boldsymbol{O(Rn)}$ ***time***. However, if we implement the priority queue as a binary heap or a heap-ordered array, we have $N = \Theta(1)$ and $I = E = O(\log n)$, so the overall algorithm runs in $\boldsymbol{O(R \log n)}$ time.

Finally, sometimes we are interested in computational resources other than time, such as space, number of coin flips, number of cache or page faults, number of inter-process messages, or the number of gifts my true love gave to me. These resources can be analyzed using the same techniques used to analyze running time. For example, lattice multiplication of two n-digit numbers requires $O(n^2)$ space if we write down all the partial products before adding them, but only $O(n)$ space if we add them on the fly.

Exercises

0. Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a trivial one-line solution!]*

♥1. (a) Identify (or write) a song that requires $\Theta(n^3)$ time to sing the first n verses.

(b) Identify (or write) a song that requires $\Theta(n \log n)$ time to sing the first n verses.

(c) Identify (or write) a song that requires some other weird amount of time to sing the first n verses.

2. Careful readers might complain that our analysis of songs like "n Bottles of Beer on the Wall" or "The n Days of Christmas" is overly simplistic, because larger numbers take longer to sing than shorter numbers. More generally, because there are only so many words of a given length, larger sets of words necessarily contain longer words.[24] We can more accurately estimate singing time by counting the number of *syllables* sung, rather than the number of *words*.

(a) How long does it take to sing the integer n?

[24] Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine!

(b) How long does it take to sing "n Bottles of Beer on the Wall"?

(c) How long does it take to sing "The n Days of Christmas"?

As usual, express your answers in the form $O(f(n))$ for some function f.

3. The cumulative drinking song "The Barley Mow" has been sung throughout the British Isles for centuries. The song has many variants; Figure 0.6 contains pseudolyrics for one version traditionally sung in Devon and Cornwall, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer.[25]

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

Figure 0.6. "The Barley Mow".

(a) Suppose each name *vessel*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. Suppose *vessel*$[n]$ has

[25] In practice, the song uses some subset of the following vessels; nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, and ocean. With a few exceptions (especially at the end), every vessel in this list has twice the volume of its predecessor. Irish and Scottish versions of the song have slightly different lyrics, and they usually switch to people (barmaid, landlord, drayer, and so on) after "gallon".

An early version of the song entitled "Give us once a drink" appears in the play *Jack Drum's Entertainment (or the Comedie of Pasquill and Katherine)* written by John Marston around 1600. ("Giue vs once a drinke for and the black bole. Sing gentle Butler *bally moy*!") There is some disagreement whether Marston wrote the "high Dutch Song" specifically for the play, whether "bally moy" is a mondegreen for "barley mow" or vice versa, or whether it's actually the same song at all. These discussions are best had over n bottles of beer.

$\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *vessel*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

4. Recall that the input to the Huntington-Hill algorithm ApportionCongress is an array $Pop[1..n]$, where $Pop[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $Rep[1..n]$, where $Rep[i]$ is the number of representatives allotted to the ith state by the algorithm.

The Huntington-Hill algorithm is sometimes described in a way that avoids the use of priority queues entirely. The top-level algorithm "guesses" a positive real number D, called the *divisor*, and then runs the following subroutine to compute an apportionment. The variable q is the ideal *quota* of representatives allocated to a state for the given divisor D; the actual number of representatives allocated is always either $\lceil q \rceil$ or $\lfloor q \rfloor$.

```
HHGuess(Pop[1..n], R, D):
  reps ← 0
  for i ← 1 to n
      q ← Pop[i]/D
      if q·q < ⌈q⌉·⌊q⌋
          Rep[i] ← ⌊q⌋
      else
          Rep[i] ← ⌈q⌉
      reps ← reps + Rep[i]
  return reps
```

There are three possibilities for the final return value *reps*. If $reps < R$, we did not allocate enough representatives, which (at least intuitively) means our divisor D was too small. If $reps > R$, we allocated too many representatives, which (at least intuitively) means our divisor D was too large. Finally, if $reps = R$, we can return the array $Rep[1..n]$ as the final apportionment. In practice, we can compute a valid apportionment (with $reps = R$) by calling HHGuess with a small number of integer divisors close to the *standard* divisor $D = P/R$.

In the following problems, let $P = \sum_{i=1}^{n} Pop[i]$ denote the total population of all n states, and assume that $n \le R \le P$.

(a) Show that calling HHGuess with the standard divisor $D = P/R$ does *not* necessarily yield a valid apportionment.

(b) Prove that if HHGuess returns the same value of *reps* for two different divisors D and D', it also computes the same allocation $Rep[1..n]$ for both of those divisors.

(c) Prove that if HHGuess returns the correct value R, it computes the same allocation $Rep[1..n]$ as our earlier algorithm ApportionCongress.

(d) Prove that a "correct" divisor D does not necessarily exist! That is, describe inputs $Pop[1..n]$ and R, where $n \le R \le P$, such that for *every* real number $D > 0$, the number of representatives allocated by HHGuess is not equal to R. *[Hint: What happens if we change $<$ to $\le$ in the fourth line of HHGuess?]*

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400BCE), translated by Lionel Giles (1910)

Our life is frittered away by detail.... Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Now, don't ask me what Voom is. I never will know.
But, boy! Let me tell you, it DOES clean up snow!

— Dr. Seuss [Theodor Seuss Geisel], *The Cat in the Hat Comes Back* (1958)

Do the hard jobs first. The easy jobs will take care of themselves.

— attributed to Dale Carnegie

1

Recursion

1.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm for X cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic.

For example, the peasant multiplication algorithm described in the previous chapter reduces the problem of multiplying two arbitrary positive integers to three simpler problems: addition, mediation (halving), and parity-checking. The algorithm relies on an abstract "positive integer" data type that supports those three operations, but the correctness of the multiplication algorithm does not

depend on the precise data representation (tally marks, clay tokens, Babylonian hexagesimal, quipu, counting rods, Roman numerals, finger positions, augrym stones, gobar numerals, binary, negabinary, Gray code, balanced ternary, phinary, quater-imaginary, ...), or on the precise implementations of those operations. Of course, the *running time* of the multiplication algorithm depends on the *running time* of the addition, mediation, and parity operations, but that's a separate issue from *correctness*. Most importantly, we can create a more efficient multiplication algorithm just by switching to a more efficient number representation (from tally marks to place-value notation, for example).

Similarly, the Huntington-Hill algorithm reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations INSERT and EXTRACTMAX. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the INSERT and EXTRACTMAX algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. That ignorance is uncomfortable for many beginners, but it is both unavoidable and extremely useful. Even when you do know precisely how your components work, it is often *extremely* helpful to pretend that you don't.

1.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more ***simpler instances of the same problem***.

If the self-reference is confusing, it may be helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the ***Recursion Fairy***. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will solve all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt*

Out.[1] Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name: the ***Induction Hypothesis***.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to simpler and simpler instances. Eventually, the recursive reductions must lead to an elementary ***base case*** that can be solved by some other method; otherwise, the recursive algorithm will loop forever. The most common way to satisfy this condition is to reduce to one or more ***smaller*** instances of the same problem. For example, if the original input is a skreeble with n glurps, the input to each recursive call should be a skreeble with strictly less than n glurps. Of course this is impossible if the skreeble has no glurps at all—You can't have negative glurps; that would be silly!—so in that case we must grindlebloff the skreeble using some other method.

We've already seen one instance of this pattern in the peasant multiplication algorithm, which is based directly on the following recursive identity.

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

The same recurrence can be expressed algorithmically as follows:

```
PeasantMultiply(x, y):
  if x = 0
      return 0
  else
      x' ← ⌊x/2⌋
      y' ← y + y
      prod ← PeasantMultiply(x', y')    ⟨⟨Recurse!⟩⟩
      if x is odd
          prod ← prod + y
      return prod
```

A lazy Egyptian scribe could execute this algorithm by computing x' and y', *asking a more junior scribe to multiply* x' *and* y', and then possibly adding y to the junior scribe's response. The junior scribe's problem is simpler because $x' < x$, and repeatedly decreasing a positive integer eventually leads to 0. How the junior scribe actually computes $x' \cdot y'$ is none of the senior scribe's business (and it's none of your business, either).

[1]When I was an undergraduate, I attributed recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone more entheogenically experienced than I might recognize these Rekursionswichtelmänner as Terence McKenna's "self-transforming machine elves".

1.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published—as an actual physical puzzle!—by the French teacher and recreational mathematician Édouard Lucas in 1883,[2] under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:[3]

> In the great temple at Benares[4]…beneath the dome which 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 discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the 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.

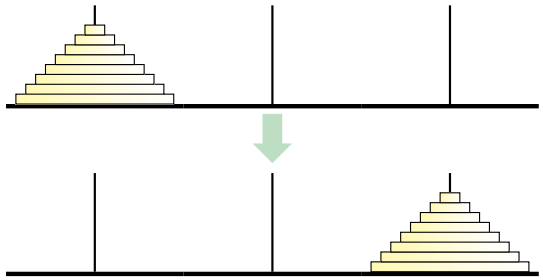


Figure 1.1. The (8-disk) Tower of Hanoi puzzle

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And because most physical instances of the puzzle are made of wood instead of diamonds and gold, I will call the three possible locations for the disks "pegs" instead of

[2]Lucas later claimed to have invented the puzzle in 1876.

[3]This English translation is taken from W. W. Rouse Ball's 1892 book *Mathematical Recreations and Essays*.

[4]The "great temple at Benares" is almost certainly the Kashi Vishvanath Temple in Varanasi, Uttar Pradesh, India, located approximately 2400km west-north-west of Hà Nội, Việt Nam, where the fictional N. Claus supposedly resided. Coincidentally, the French Army invaded Hanoi in 1883, the same year Lucas released his puzzle, ultimately leading to its establishment as the capital of French Indochina.

"needles". How can we move a tower of n disks from one peg to another, using a third spare peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

As N. Claus (de Siam) pointed out in the pamphlet included with his puzzle, the secret to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are in the way. So first we have to move those $n-1$ smaller disks to the spare peg. Once that's done, we can move the largest disk directly to its destination. Finally, to finish the puzzle, we have to move the $n-1$ smaller disks from the spare peg to their destination.

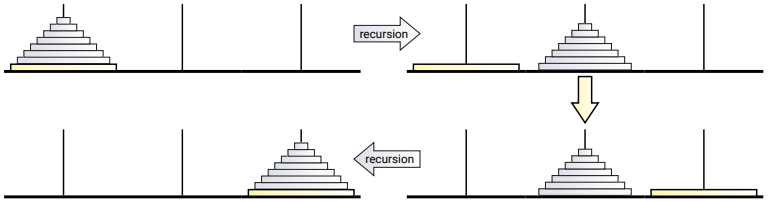


Figure 1.2. The Tower of Hanoi algorithm; ignore everything but the bottom disk.

So now all we have to figure out is how to—

NO!! STOP!!

That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy—or to carry Lucas's metaphor further, to the junior monks at the temple. *Our* job is finished. If we didn't trust the junior monks, we wouldn't have hired them; let them do their job in peace.

Our reduction does make one subtle but extremely important assumption: *There is a largest disk*. Our recursive algorithm works for any *positive* number of disks, but it breaks down when $n = 0$. We must handle that case using a different method. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all, by doing nothing.

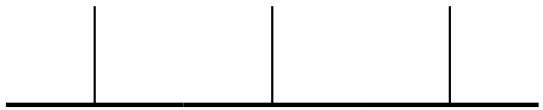


Figure 1.3. The vacuous base case for the Tower of Hanoi algorithm. There is no spoon.

It may be tempting to think about how all those smaller disks move around—or more generally, what happens when the recursion is unrolled—but really, don't do it. For most recursive algorithms, unrolling the recursion is neither necessary nor helpful. Our ***only*** task is to reduce the problem instance we're given to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our recursive Tower of Hanoi algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves the top $n-1$ disks (more formally, the Inductive Hypothesis implies that our recursive algorithm correctly moves the top $n-1$ disks) so our algorithm is correct.

The recursive Hanoi algorithm is expressed in pseudocode in Figure 1.4. The algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder. Notice that the algorithm correctly does nothing at all when $n = 0$.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)    ⟨⟨Recurse!⟩⟩
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)    ⟨⟨Recurse!⟩⟩
```

Figure 1.4. A recursive algorithm to solve the Tower of Hanoi

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. By writing out the first several values of $T(n)$, we can easily guess that $\boldsymbol{T(n) = 2^n - 1}$; a straightforward induction proof implies that this guess is correct. In particular, moving a tower of 64 disks requires $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$ individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years ("plus de *cinq milliards de siècles*") before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.4 Mergesort

Mergesort is one of the earliest algorithms designed for general-purpose stored-program computers. The algorithm was developed by John von Neumann in 1945, and described in detail in a publication with Herman Goldstine in 1947, as one of the first non-numerical programs for the EDVAC.[5]

[5] Goldstine and von Neumann actually described an non-recursive variant now usually called bottom-up mergesort. At the time, large data sets were sorted by special-purpose machines—almost all built by IBM—that manipulated punched cards using variants of binary radix sort. Von

1. Divide the input array into two subarrays of roughly equal size.
2. Recursively mergesort each of the subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.5. A mergesort example.

The first step is completely trivial—just divide the array size by two—and we can delegate the second step to the Recursion Fairy. All the real work is done in the final merge step. A complete description of the algorithm is given in Figure 1.6; to keep the recursive structure clear, I've extracted the merge step into an independent subroutine. The merge algorithm is also recursive—identify the first element of the output array, and then recursively merge the rest of the input arrays.

```
MergeSort(A[1..n]):
  if n > 1
    m ← ⌊n/2⌋
    MergeSort(A[1..m])        ⟨⟨Recurse!⟩⟩
    MergeSort(A[m+1..n])      ⟨⟨Recurse!⟩⟩
    Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i];  i ← i+1
    else if i > m
      B[k] ← A[j];  j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i];  i ← i+1
    else
      B[k] ← A[j];  j ← j+1
  for k ← 1 to n
    A[k] ← B[k]
```

Figure 1.6. Mergesort

Correctness

To prove that this algorithm is correct, we apply our old friend induction twice, first to the Merge subroutine then to the top-level MergeSort algorithm.

Lemma 1.1. *Merge correctly merges the subarrays $A[1..m]$ and $A[m+1..n]$, assuming those subarrays are sorted in the input.*

Neumann argued (successfully!) that because the EDVAC could sort faster than IBM's dedicated sorters, "without human intervention or need for additional equipment", the EDVAC was an "all purpose" machine, and special-purpose sorting machines were no longer necessary.

Proof: Let $A[1..n]$ be any array and m any integer such that the subarrays $A[1..m]$ and $A[m+1..n]$ are sorted. We prove that for all k from 0 to n, the last $n-k-1$ iterations of the main loop correctly merge $A[i..m]$ and $A[j..n]$ into $B[k..n]$. The proof proceeds by induction on $n-k+1$, the number of elements remaining to be merged.

If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.) Otherwise, there are four cases to consider for the kth iteration of the main loop.

- If $j > n$, then subarray $A[j..n]$ is empty, so $\min\big(A[i..m] \cup A[j..n]\big) = A[i]$.
- If $i > m$, then subarray $A[i..m]$ is empty, so $\min\big(A[i..m] \cup A[j..n]\big) = A[j]$.
- Otherwise, if $A[i] < A[j]$, then $\min\big(A[i..m] \cup A[j..n]\big) = A[i]$.
- Otherwise, we must have $A[i] \geq A[j]$, and $\min\big(A[i..m] \cup A[j..n]\big) = A[j]$.

In all four cases, $B[k]$ is correctly assigned the smallest element of $A[i..m] \cup A[j..n]$. In the two cases with the assignment $B[k] \leftarrow A[i]$, the Recursion Fairy correctly merges—sorry, I mean the Induction Hypothesis implies that the last $n-k$ iterations of the main loop correctly merge $A[i+1..m]$ and $A[j..n]$ into $B[k+1..n]$. Similarly, in the other two cases, the Recursion Fairy also correctly merges the rest of the subarrays. □

Theorem 1.2. *MergeSort correctly sorts any input array $A[1..n]$.*

Proof: We prove the theorem by induction on n. If $n \leq 1$, the algorithm correctly does nothing. Otherwise, the Recursion Fairy correctly sorts—sorry, I mean the induction hypothesis implies that our algorithm correctly sorts the two smaller subarrays $A[1..m]$ and $A[m+1..n]$, after which they are correctly Merged into a single sorted array (by Lemma 1.1). □

Analysis

Because the MergeSort algorithm is recursive, its running time is naturally expressed as a recurrence. Merge clearly takes $O(n)$ time, because it's a simple for-loop with constant work per iteration. We immediately obtain the following recurrence for MergeSort:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

As in most divide-and-conquer recurrences, we can safely strip out the floors and ceilings (using a technique called *domain transformations* described later in this chapter), giving us the simpler recurrence $T(n) = 2T(n/2) + O(n)$. The "all levels equal" case of the recursion tree method (also described later in this chapter) immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. Even if you are not (yet) familiar with recursion trees, you can verify the solution $T(n) = O(n \log n)$ by induction.

1.5 Quicksort

Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1959 and first published in 1961. In this algorithm, the hard work is splitting the array into smaller subarrays *before* recursion, so that merging the sorted subarrays is trivial.

1. Choose a *pivot* element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.
3. Recursively quicksort the first and last subarrays.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	O	E	I	N	L	M	P	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	P	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.7. A quicksort example.

More detailed pseudocode is given in Figure 1.8. In the PARTITION subroutine, the input parameter p is the index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot element. There are many different efficient partitioning algorithms; the one I'm presenting here is attributed to Nico Lomuto.[6] The variable ℓ counts the number of items in the array that are ℓess than the pivot element.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r−1])    ⟨⟨Recurse!⟩⟩
      QUICKSORT(A[r+1..n])    ⟨⟨Recurse!⟩⟩
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                       ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n−1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ+1]
  return ℓ + 1
```

Figure 1.8. Quicksort

Correctness

Just like mergesort, proving that QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and the other to prove that QUICKSORT correctly sorts *assuming* PARTITION is correct. To prove PARTITION is correct, we need to prove the following loop invariant: At the end of each iteration of the main loop, everything in the subarray $A[1..\ell]$ is ℓess than $A[n]$, and nothing in the subarray $A[\ell+1..i]$ is less than $A[n]$. I'll leave the remaining straightforward but tedious details as exercises for the reader.

[6] Hoare proposed a more complicated "two-way" partitioning algorithm that has some practical advantages over Lomuto's algorithm. On the other hand, Hoare's partitioning algorithm is one of the places off-by-one errors go to die.

Analysis

The analysis of quicksort is also similar to that of mergesort. PARTITION clearly runs in $O(n)$ time, because it's a simple for-loop with constant work per iteration. For QUICKSORT, we get a recurrence that depends on r, the *rank* of the chosen pivot element:

$$T(n) = T(r-1) + T(n-r) + O(n)$$

If we could somehow always magically choose the pivot to be the *median* element of the array A, we would have $r = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) \;=\; T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \;\le\; 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ using either the recursion tree method or the even simpler "Oh yeah, we already solved that recurrence for mergesort" method.

In fact, as we will see later in this chapter, we *can* actually locate the median element in an unsorted array in linear time, but the algorithm is fairly complicated, and the hidden constant in the $O(\cdot)$ notation is large enough to make the resulting sorting algorithm impractical. In practice, most programmers settle for something simple, like choosing the first or last element of the array. In this case, r can take any value between 1 and n, so we have

$$T(n) = \max_{1 \le r \le n} \big(T(r-1) + T(n-r) + O(n) \big).$$

In the worst case, the two subproblems are completely unbalanced—either $r = 1$ or $r = n$—and the recurrence becomes $T(n) \le T(n-1) + O(n)$. The solution is $\boldsymbol{T(n) = O(n^2)}$.

Another common heuristic is called "median of three"—choose three elements (usually at the beginning, middle, and end of the array), and take the median of those three elements as the pivot. Although this heuristic is somewhat more efficient in practice than just choosing one element, especially when the array is already (nearly) sorted, we can still have $r = 2$ or $r = n-1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \le T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element should "usually" fall somewhere in the middle of the array, say with rank between $n/10$ and $9n/10$. This observation suggests that the "average-case" running time should be $O(n \log n)$. Although this intuition can be formalized, the most common formalization makes the completely unrealistic assumption that all permutations of the input array are equally likely. Real world data *may* be random, but it is not random in any way that we can predict in advance, and it is *certainly* not uniform![7]

Occasionally people also consider "best case" running time for some reason. We won't.

1.6 The Pattern

Both mergesort and quicksort follow a general three-step pattern called ***divide and conquer***:

1. **Divide** the given instance of the problem into several *independent smaller* instances of *exactly* the same problem.
2. **Delegate** each smaller instance to the Recursion Fairy.
3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

If the size of any instance falls below some constant threshold, we abandon recursion and solve the problem directly, by brute force, in constant time.

Proving a divide-and-conquer algorithm correct almost always requires induction. Analyzing the running time requires setting up and solving a recurrence, which usually (but unfortunately not always!) can be solved using recursion trees.

1.7 Recursion Trees

So what are these "recursion trees" I keep talking about? Recursion trees are a simple, general, pictorial tool for solving divide-and-conquer recurrences. A recursion tree is a rooted tree with one node for each recursive subproblem. The *value* of each node is the amount of time spent on the corresponding subproblem *excluding* recursive calls. Thus, the overall running time of the algorithm is the sum of the values of all nodes in the tree.

To make this idea more concrete, imagine a divide-and-conquer algorithm that spends $O(f(n))$ time on non-recursive work, and then makes r recursive

[7] On the other hand, if we choose the pivot index p uniformly at random, then QUICKSORT runs in $O(n \log n)$ time *with high probability*, for *every* possible input array. The key difference is that the randomness is controlled by our algorithm, not by the All-Powerful Malicious Adversary who gives us input data after reading our code. The analysis of randomized quicksort is unfortunately outside the scope of this book, but you can find relevant lecture notes at http://algorithms.wtf/.

calls, each on a problem of size n/c. Up to constant factors (which we can hide in the $O(\,)$ notation), the running time of this algorithm is governed by the recurrence

$$T(n) = r\,T(n/c) + f(n).$$

The root of the recursion tree for $T(n)$ has value $f(n)$ and r children, each of which is the root of a (recursively defined) recursion tree for $T(n/c)$. Equivalently, a recursion tree is a complete r-ary tree where each node at depth d contains the value $f(n/c^d)$. (Feel free to assume that n is an integer power of c, so that n/c^d is always an integer, although in fact this doesn't matter.)

In practice, I recommend drawing out the first two or three levels of the tree, as in Figure 1.9.

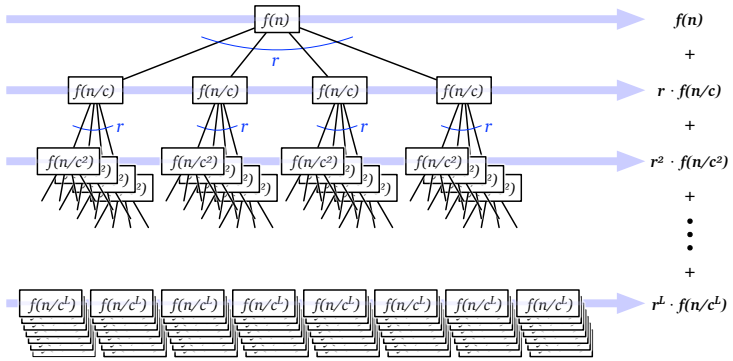


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

The leaves of the recursion tree correspond to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the precise base case doesn't actually matter; we can safely assume $T(n) = 1$ for all $n \le n_0$, where n_0 is an arbitrary positive constant. In particular, we can choose whatever value of n_0 is most convenient for our analysis. For this example, I'll choose $n_0 = 1$.

Now $T(n)$ is the sum of all values in the recursion tree; we can evaluate this sum by considering the tree level-by-level. For each integer i, the ith level of the tree has exactly r^i nodes, each with value $f(n/c^i)$. Thus,

$$T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i) \qquad (\Sigma)$$

where L is the depth of the tree. Our base case $n_0 = 1$ immediately implies $\boldsymbol{L = \log_c n}$, because $n/c^L = n_0 = 1$. It follows that the number of leaves in

the recursion tree is exactly $r^L = r^{\log_c n} = n^{\log_c r}$. Thus, the last term in the level-by-level sum (Σ) is $n^{\log_c r} \cdot f(1) = O(n^{\log_c r})$, because $f(1) = O(1)$.

There are three common cases where the level-by-level series (Σ) is especially easy to evaluate:

- **Decreasing:** If the series *decays exponentially*—every term is a constant factor smaller than the previous term—then $T(n) = O(f(n))$. In this case, the sum is dominated by the value at the root of the recursion tree.
- **Equal:** If all terms in the series are equal, we immediately have $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$. (The constant c vanishes into the $O(\,)$ notation.)
- **Increasing:** If the series *grows exponentially*—every term is a constant factor larger than the previous term—then $T(n) = O(n^{\log_c r})$. In this case, the sum is dominated by the number of leaves in the recursion tree.

In the first and third cases, only the largest term in the geometric series matters; all other terms are swallowed up by the $O(\cdot)$ notation. In the decreasing case, we don't even have to compute L; the asymptotic upper bound would still hold if the recursion tree were infinite!

As an elementary example, if we draw out the first few levels of the recursion tree for the (simplified) mergesort recurrence $T(n) = 2T(n/2) + O(n)$, we discover that all levels are equal, which immediately implies $T(n) = O(n \log n)$.

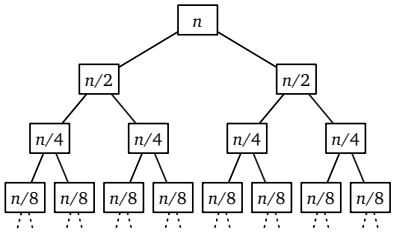


Figure 1.10. The recursion tree for mergesort

The recursion tree technique can also be used for algorithms where the recursive subproblems have different sizes. For example, if we could somehow implement quicksort so that the pivot *always* lands in the middle third of the sorted array, the worst-case running time would satisfy the recurrence

$$T(n) \le T(n/3) + T(2n/3) + O(n).$$

This recurrence might look scary, but it's actually pretty tame. If we draw out a few levels of the resulting recursion tree, we quickly realize that the sum of values on any level is *at most* n—deeper levels might be missing some nodes—and the entire tree has depth $\log_{3/2} n = O(\log n)$. It immediately follows that $T(n) = O(n \log n)$. (Moreover, the number of *full* levels in the recursion

tree is $\log_3 n = \Omega(\log n)$, so this conservative analysis can be improved by at most a constant factor, which for our purposes means not at all.) The fact that the recursion tree is unbalanced simply doesn't matter.

As a more extreme example, the worst-case recurrence for quicksort $T(n) = T(n-1) + T(1) + O(n)$ gives us a completely unbalanced recursion tree, where one child of each internal node is a leaf. The level-by-level sum doesn't fall into any of our three default categories, but we can still derive the solution $T(n) = O(n^2)$ by observing that every level value is at most n and there are at most n levels. (Again, this conservative analysis is tight, because $n/2$ levels each have value at least $n/2$.)

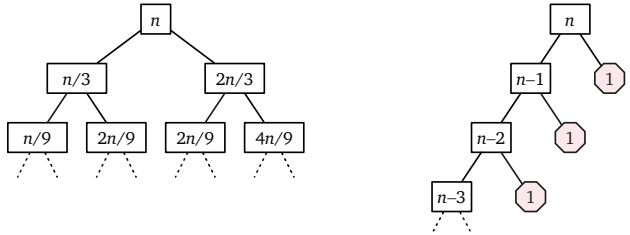


Figure 1.11. Recursion trees for quicksort with good pivots (left) and with worst-case pivots (right)

♥ Ignoring Floors and Ceilings Is Okay, Honest

Careful readers might object that our analysis brushes an important detail under the rug. The running time of mergesort doesn't *really* obey the recurrence $T(n) = 2T(n/2) + O(n)$; after all, the input size n might be odd, and what could it possibly mean to sort an array of size $42\frac{1}{2}$ or $17\frac{7}{8}$? The actual mergesort recurrence is somewhat messier:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

Sure, we could check that $T(n) = O(n \log n)$ using induction, but the necessary calculations would be awful. Fortunately, there is a simple technique for removing floors and ceilings from recurrences, called *domain transformation*.

- First, because we are deriving an upper bound, we can safely overestimate $T(n)$, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:[8]

$$T(n) \le 2T\big(\lceil n/2 \rceil\big) + n \le 2T(n/2 + 1) + n.$$

[8] Formally, we are treating T as a function over the *reals*, not just over the integers, that satisfies the given recurrence with the base case $T(n) = C$ for all $n \le n_0$, for some real numbers $C \ge 0$ and $n_0 > 0$ whose values don't matter. If n happens to be an integer, then $T(n)$ coincides with the running time of an algorithm on an input of size n, but that doesn't matter, either.

- Second, we define a new function $S(n) = T(n+\alpha)$, choosing the constant α so that $S(n)$ satisfies the simpler recurrence $S(n) \le 2S(n/2) + O(n)$. To find the correct constant α, we derive a recurrence for S from our given recurrence for T:

$$\begin{aligned} S(n) &= T(n+\alpha) && \text{[definition of } S\text{]} \\ &\le 2T(n/2+\alpha/2+1)+n+\alpha && \text{[recurrence for } T\text{]} \\ &= 2S(n/2-\alpha/2+1)+n+\alpha && \text{[definition of } S\text{]} \end{aligned}$$

  Setting $\alpha = 2$ simplifies this recurrence to $S(n) \le 2S(n/2) + n + 2$, which is exactly what we wanted.

- Finally, the recursion tree method implies $S(n) = O(n \log n)$, and therefore

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n),$$

  exactly as promised.

Similar domain transformations can be used to remove floors, ceilings, and even lower order terms from *any* divide and conquer recurrence. But now that we realize this, we don't need to bother grinding through the details ever again! From now on, faced with any divide-and-conquer recurrence, I will silently brush floors and ceilings and lower-order terms under the rug, and I encourage you to do the same.

♥1.8 Linear-Time Selection

During our discussion of quicksort, I claimed in passing that we can find the median of an unsorted array in linear time. The first such algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth smallest element in an n-element array, given the array and the integer k as input, using a variant of an algorithm called *quickselect* or *one-armed quicksort*. Quickselect was first described by Tony Hoare in 1961, literally on the same page where he first published quicksort.

Quickselect

The generic quickselect algorithm chooses a pivot element, partitions the array using the same PARTITION subroutine as QUICKSORT, and then recursively searches *only one* of the two subarrays, specifically, the one that contains the kth smallest element of the original input array. Pseudocode for quickselect is given in Figure 1.12.


```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)
      if k < r
          return QUICKSELECT(A[1..r−1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k−r)
      else
          return A[r]
```

Figure 1.12. Quickselect, or one-armed quicksort

This algorithm has two important features. First, just like quicksort, the correctness of quickselect does not depend on how the pivot is chosen. Second, even if we really only care about selecting *medians* (the special case $k = n/2$), Hoare's recursive strategy requires us to consider the more general selection problem; the median of the input array $A[1..n]$ is almost never the median of either of the two smaller subarrays $A[1..r-1]$ or $A[r+1..n]$.

The worst-case running time of QUICKSELECT obeys a recurrence similar to QUICKSORT. We don't know the value of r, or which of the two subarrays we'll recursively search, so we have to assume the worst.

$$T(n) \le \max_{1 \le r \le n} \max\{T(r-1), T(n-r)\} + O(n)$$

We can simplify the recurrence slightly by letting ℓ denote the length of the recursive subproblem:

$$T(n) \le \max_{0 \le \ell \le n-1} T(\ell) + O(n)$$

If the chosen pivot element is always either the smallest or largest element in the array, the recurrence simplifies to $T(n) = T(n-1) + O(n)$, which implies $T(n) = O(n^2)$. (The recursion tree for this recurrence is just a simple path.)

Good pivots

We could avoid this quadratic worst-case behavior if we could *somehow* magically choose a ***good*** pivot, meaning $\ell \le \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \le T(\alpha n) + O(n).$$

This recurrence expands into a decreasing geometric series, which is dominated by its largest term, so $T(n) = O(n)$. (Again, the recursion tree is just a simple path. The constant in the $O(n)$ running time depends on the constant α.)

In other words, if we could somehow quickly find an element that's even *close* to the median in linear time, we could find the *exact* median in linear time. So now all we need is an Approximate Median Fairy. The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good quickselect pivot by ***recursively*** computing the median of a carefully-chosen subset of the input array. The Approximate Median Fairy is just the Recursion Fairy in disguise!

Specifically, we divide the input array into $\lceil n/5 \rceil$ *blocks*, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We compute the median of each block by brute force, collect those medians into a new array $M[1..\lceil n/5 \rceil]$, and then *recursively* compute the median of this new array. Finally, we use the median of the block medians (called "*mom*" in the pseudocode below) as the quickselect pivot.

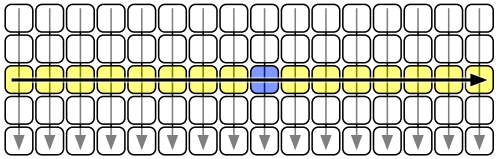
```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i − 4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MomSelect(M[1..m], ⌊m/2⌋)          ⟨⟨Recursion!⟩⟩

      r ← Partition(A[1..n], mom)

      if k < r
          return MomSelect(A[1..r − 1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MomSelect(A[r + 1..n], k − r) ⟨⟨Recursion!⟩⟩
      else
          return mom
```

MomSelect uses recursion for two different purposes; the first time to choose a pivot element (*mom*), and the second time to search through the entries on one side of that pivot.

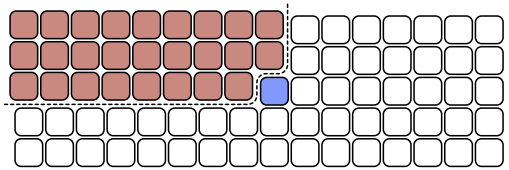
Analysis

But why is this fast? The first key insight is that ***the median of medians is a good pivot***. Mom is larger than $\lfloor \lceil n/5 \rceil / 2 \rfloor - 1 \approx n/10$ block medians, and each block median is larger than two other elements in its block. Thus, mom is bigger than at least $3n/10$ elements in the input array; symmetrically, mom is smaller than at least $3n/10$ elements. Thus, in the worst case, the second recursive call searches an array of size at most $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not actually do this!*) In this arrangement, the median-of-medians is the element closest to the center of the grid.



The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than mom. If the element we're looking for is larger than mom, our algorithm will throw away *everything* smaller than mom, including those $3n/10$ elements, before recursing. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. A symmetric argument implies that if our target element is smaller than mom, we discard at least $3n/10$ elements larger than mom, so the input to our recursive subproblem has at most $7n/10$ elements.



Okay, so mom is a good pivot, but our algorithm still makes *two* recursive calls instead of just one; how do we prove linear time? The second key insight is that the *total* size of the two recursive subproblems is a constant factor smaller than the size of the original input array. The worst-case running time of the algorithm obeys the recurrence

$$T(n) \le T(n/5) + T(7n/10) + O(n).$$

If we draw out the recursion tree for this recurrence, we observe that the total work at each level of the recursion tree is at most $9/10$ the total work at the previous level. Thus, the level sums decay exponentially, giving us the solution $T(n) = O(n)$. (Again, the fact that the recursion tree is unbalanced is completely immaterial.) Hooray! Thanks, Mom!

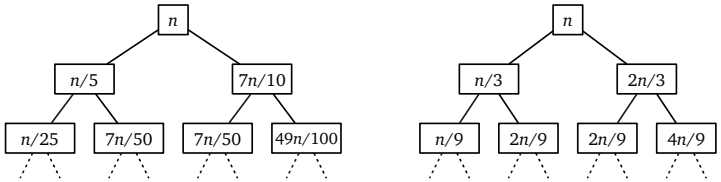


Figure 1.13. The recursion trees for MomSelect and a similar selection algorithm with blocks of size 3

Sanity Checking

At this point, many students ask about that magic constant 5. Why did we choose that particular block size? The answer is that 5 is the smallest odd block size that gives us exponential decay in the recursion-tree analysis! (Even block sizes introduce additional complications.) If we had used blocks of size 3 instead, the running-time recurrence would be

$$T(n) \le T(n/3) + T(2n/3) + O(n).$$

We've seen this recurrence before! Every level of the recursion tree has total value *at most* n, and the depth of the recursion tree is $\log_{3/2} n = O(\log n)$, so the solution to this recurrence is $T(n) \le O(n \log n)$. (Moreover, this analysis is tight, because the recursion tree has $\log_3 n$ complete levels.) Median-of-medians selection using 3-element blocks is no faster than sorting.

Finer analysis reveals that the constant hidden by the $O(\,)$ notation is quite large, even if we count only comparisons. Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. Naïvely partitioning the array after the recursive call would require $n-1$ comparisons, but we already know $3n/10$ elements larger than the pivot and $3n/10$ elements smaller than the pivot, so partitioning actually requires only $2n/5$ additional comparisons. Thus, a more precise recurrence for the worst-case number of comparisons is

$$T(n) \le T(n/5) + T(7n/10) + 8n/5.$$

The recursion tree method implies the upper bound

$$T(n) \le \frac{8n}{5} \sum_{i \ge 0} \left(\frac{9}{10}\right)^i = \frac{8n}{5} \cdot 10 = 16n.$$

In practice, median-of-medians selection is not as slow as this worst-case analysis predicts—getting a worst-case pivot at every level of recursion is incredibly unlikely—but it is still slower than sorting for even moderately large arrays.[9]

[9] In fact, the right way to choose the pivot element in practice is to choose it *uniformly at random*. Then the expected number of comparisons required to find the median is at most $4n$. See my randomized algorithms lecture notes at http://algorithms.wtf for more details.

1.9 Fast Multiplication

In the previous chapter, we saw two ancient algorithms for multiplying two n-digit numbers in $O(n^2)$ time: the grade-school lattice algorithm and the Egyptian peasant algorithm.

Maybe we can get a more efficient algorithm by splitting the digit arrays in half and exploiting the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

This recurrence immediately suggests the following divide-and-conquer algorithm to multiply two n-digit numbers x and y. Each of the four sub-products ac, bc, ad, and bd is computed recursively, but the multiplications in the last line are *not* recursive, because we can multiply by a power of ten by shifting the digits to the left and filling in the correct number of zeros, all in $O(n)$ time.

```
SplitMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← SplitMultiply(a, c, m)
      f ← SplitMultiply(b, d, m)
      g ← SplitMultiply(b, c, m)
      h ← SplitMultiply(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

Correctness of this algorithm follows easily by induction. The running time for this algorithm follows the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n).$$

The recursion tree method transforms this recurrence into an *increasing* geometric series, which implies $T(n) = O(n^{\log_2 4}) = O(n^2)$. In fact, this algorithm multiplies each digit of x with each digit of y, just like the lattice algorithm. So I guess that didn't work. Too bad. It was a nice idea.

In the mid-1950s, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that there is *no* algorithm to multiply two n-digit numbers in subquadratic time. Kolmogorov organized a seminar at Moscow University in 1960, where he restated his "n^2 conjecture" and posed several related problems that he planned to discuss at future meetings. Almost exactly a week later, a 23-year-old student named Anatoliĭ Karatsuba presented Kolmogorov with a remarkable counterexample. According to Karatsuba himself,

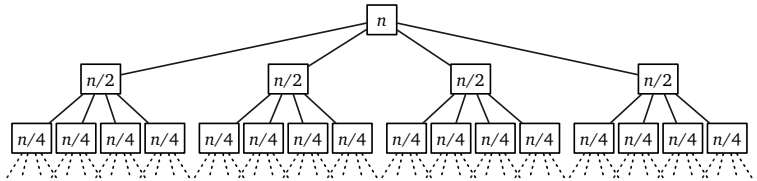


Figure 1.14. The recursion tree for naïve divide-and-conquer multiplication

> After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

Karatsuba observed that the middle coefficient $bc+ad$ can be computed from the other two coefficients ac and bd using only *one* more recursive multiplication, via the following algebraic identity:

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

This trick lets us replace the four recursive calls in the previous algorithm with only three recursive calls, as shown below:

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

The running time of Karatsuba's FastMultiply algorithm follows the recurrence

$$T(n) \le 3T(\lceil n/2 \rceil) + O(n)$$

Once again, the recursion tree method transforms this recurrence into an increasing geometric series, but the new solution is only $T(n) = O(n^{\log_2 3}) = \boldsymbol{O(n^{1.58496})}$, a significant improvement over our earlier quadratic time bound.[10]

[10] My presentation simplifies the actual history slightly. In fact, Karatsuba proposed an algorithm based on the formula $(a+b)(c+d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is slightly messier: $a-b$ and $c-d$ are still m-digit numbers, but $a+b$ and $c+d$ might each have $m+1$ digits. The simplification presented here is due to Donald Knuth.

Karatsuba's algorithm arguably launched the design and analysis of algorithms as a formal field of study.

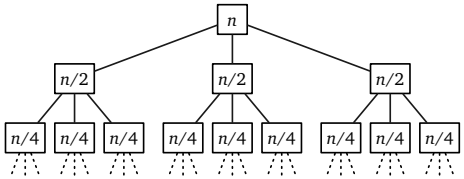


Figure 1.15. The recursion tree for Karatsuba's divide-and-conquer multiplication algorithm

We can take Karatsuba's idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to obtain even faster multiplication algorithms. Andrei Toom discovered an infinite family of algorithms that split any integer into k parts, each with n/k digits, and then compute the product using only $2k-1$ recursive multiplications; Toom's algorithms were further simplified by Stephen Cook in his PhD thesis. For any fixed k, the Toom-Cook algorithm runs in $O(n^{1+1/(\lg k)})$ time, where the hidden constant in the $O(\cdot)$ notation depends on k.

Ultimately, this divide-and-conquer strategy led Gauss (yes, really) to the discovery of the ***Fast Fourier transform***.[11] The basic FFT algorithm itself runs in $O(n \log n)$ time; however, using FFTs for integer multiplication incurs some small additional overhead. The first FFT-based integer multiplication algorithm, published by Arnold Schönhage and Volker Strassen in 1971, runs in $O(n \log n \log\log n)$ time. Schönhage-Strassen remained the theoretically fastest integer multiplication algorithm for several decades, before Martin Fürer discovered the first of a long series of technical improvements. Finally, in 2019, David Harvey and Joris van der Hoeven published an algorithm that runs in $O(n \log n)$ time.[12]

1.10 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that performs $n-1$ multiplications by a:

[11] See http://algorithms.wtf for lecture notes on Fast Fourier transforms.

[12] Schönhage-Strassen is actually the fastest algorithm *in practice* for multiplying integers with more than about 75000 digits; the more recent algorithms of Fürer, Harvey, van der Hoeven, and others would be faster "in practice" only for integers with more digits than there are particles in the universe.

```
SlowPower(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

This iterative algorithm requires n multiplications.

The input parameter a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). Because we don't know what kind of object we're multiplying, we *can't* know how much time a single multiplication requires, so we're forced to analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, originally proposed by the Indian prosodist Piṅgala in the 2nd century BCE, which uses the following simple recursive formula:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
PiṅgalaPower(a, n):
  if n = 1
      return a
  else
      x ← PiṅgalaPower(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

The total number of multiplications performed by this algorithm satisfies the recurrence $T(n) \le T(n/2) + 2$. The recursion-tree method immediately give us the solution $T(n) = O(\log n)$.

A nearly identical exponentiation algorithm can also be derived directly from the Egyptian peasant multiplication algorithm from the previous chapter, by replacing addition with multiplication (and in particular, replacing duplation with squaring).

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{otherwise} \end{cases}$$


```
PeasantPower(a, n):
  if n = 1
      return a
  else if n is even
      return PeasantPower(a², n/2)
  else
      return PeasantPower(a², ⌊n/2⌋) · a
```

This algorithm—which might reasonably be called "squaring and mediation"—also performs only $O(\log n)$ multiplications.

Both of these algorithms are asymptotically optimal; any algorithm that computes a^n *must* perform at least $\Omega(\log n)$ multiplications, because each multiplication at most doubles the largest power computed so far. In fact, when n is a power of two, both of these algorithms require exactly $\log_2 n$ multiplications, which is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, PiṅgalaPower and PeasantPower each compute a^{15} using six multiplications, but in fact only five multiplications are necessary:

- Piṅgala: $a \to a^2 \to a^3 \to a^6 \to a^7 \to a^{14} \to a^{15}$
- Peasant: $a \to a^2 \to a^4 \to a^8 \to a^{12} \to a^{14} \to a^{15}$
- Optimal: $a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$

It is a long-standing open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

Exercises

Tower of Hanoi

1. Prove that the original recursive Tower of Hanoi algorithm performs *exactly* the same sequence of moves—the same disks, to and from the same pegs, in the same order—as each of the following non-recursive algorithms. The pegs are labeled 0, 1, and 2, and our problem is to move a stack of n disks from peg 0 to peg 2 (as shown on page 24).

   (a) If n is even, swap pegs 1 and 2. At the ith step, make the only legal move that avoids peg i mod 3. If there is no legal move, then all disks are on peg i mod 3, and the puzzle is solved.

   (b) For the first move, move disk 1 to peg 1 if n is even and to peg 2 if n is odd. Then repeatedly make the only legal move that involves a different disk from the previous move. If no such move exists, the puzzle is solved.

   (c) Pretend that disks $n+1$, $n+2$, and $n+3$ are at the bottom of pegs 0, 1, and 2, respectively. Repeatedly make the only legal move that satisfies the following constraints, until no such move is possible.

- Do not place an odd disk directly on top of another odd disk.
- Do not place an even disk directly on top of another even disk.
- Do not undo the previous move.

(d) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*.

```
RulerHanoi(n):
  i ← 1
  while ρ(i) ≤ n
      if n − i is even
          move disk ρ(i) forward      ⟨⟨0 → 1 → 2 → 0⟩⟩
      else
          move disk ρ(i) backward     ⟨⟨0 → 2 → 1 → 0⟩⟩
      i ← i + 1
```

2. The Tower of Hanoi is a relatively recent descendant of a much older mechanical puzzle known as the Chinese linked rings, Baguenaudier, Cardan's Rings, Meleda, Patience, Tiring Irons, Prisoner's Lock, Spin-Out, and many other names. This puzzle was already well known in both China and Europe by the 16th century. The Italian mathematician Luca Pacioli described the 7-ring puzzle and its solution in his unpublished treatise *De Viribus Quantitatis*, written between 1498 and 1506;[13] only a few years later, the Ming-dynasty poet Yang Shen described the 9-ring puzzle as "a toy for women and children". The puzzle is apocryphally attributed to a 2nd-century Chinese general, who gave the puzzle to his wife to occupy her time while he was away at war.

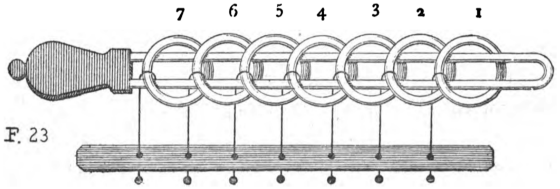


Figure 1.16. The 7-ring Baguenaudier, from *Récréations Mathématiques* by Édouard Lucas (1891) (See Image Credits at the end of the book.)

[13] *De Viribus Quantitatis* [*On the Powers of Numbers*] is an important early work on recreational mathematics and perhaps the oldest surviving treatise on magic. Pacioli is better known for *Summa de Aritmetica*, a near-complete encyclopedia of late 15th-century mathematics, which included the first description of double-entry bookkeeping.

The Baguenaudier puzzle has many physical forms, but one of the most common consists of a long metal loop and several rings, which are connected to a solid base by movable rods. The loop is initially threaded through the rings as shown in Figure 1.16; the goal of the puzzle is to remove the loop.

More abstractly, we can model the puzzle as a sequence of bits, one for each ring, where the ith bit is 1 if the loop passes through the ith ring and 0 otherwise. (Here we index the rings from right to left, as shown in Figure 1.16.) The puzzle allows two legal moves:

- You can always flip the 1st (= rightmost) bit.
- If the bit string ends with exactly z 0s, you can flip the $(z+2)$th bit.

The goal of the puzzle is to transform a string of n 1s into a string of n 0s. For example, the following sequence of 21 moves solves the 5-ring puzzle:

$$\begin{aligned}
11111 &\xrightarrow{1} 11110 \xrightarrow{3} 11010 \xrightarrow{1} 11011 \xrightarrow{2} 11001 \xrightarrow{1} 11000 \\
&\xrightarrow{5} 01000 \xrightarrow{1} 01001 \xrightarrow{2} 01011 \xrightarrow{1} 01010 \xrightarrow{3} 01110 \\
&\xrightarrow{1} 01111 \xrightarrow{2} 01101 \xrightarrow{1} 01100 \xrightarrow{4} 00100 \xrightarrow{1} 00101 \\
&\xrightarrow{2} 00111 \xrightarrow{1} 00110 \xrightarrow{3} 00010 \xrightarrow{1} 00011 \xrightarrow{2} 00001 \xrightarrow{1} 00000
\end{aligned}$$

♦(a) Call a sequence of moves *reduced* if no move is the inverse of the previous move. Prove that for any non-negative integer n, there is *exactly one* reduced sequence of moves that solves the n-ring Baguenaudier puzzle. *[Hint: This problem is much easier if you're already familiar with graphs.]*

(b) Describe an algorithm to solve the Baguenaudier puzzle. Your input is the number of rings n; your algorithm should print a reduced sequence of moves that solves the puzzle. For example, given the integer 5 as input, your algorithm should print the sequence 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1.

(c) *Exactly* how many moves does your algorithm perform, as a function of n? Prove your answer is correct.

3. A less familiar chapter in the Tower of Hanoi's history is its brief relocation of the temple from Benares to Pisa in the early 13th century.[14] The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula.

[14] Portions of this story are actually true.

Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.

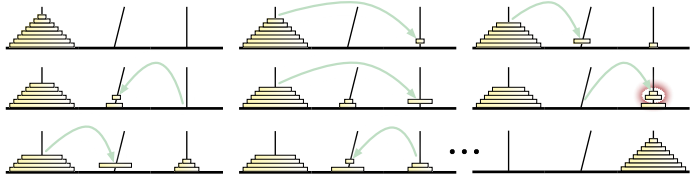


Figure 1.17. The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa than they could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

4. Consider the following restricted variants of the Tower of Hanoi puzzle In each problem, the pegs are numbered 0, 1, and 2, and your task is to move a stack of n disks from peg 0 to peg 2, exactly as in problem 1.

   (a) Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

   ♣♥(b) Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make?

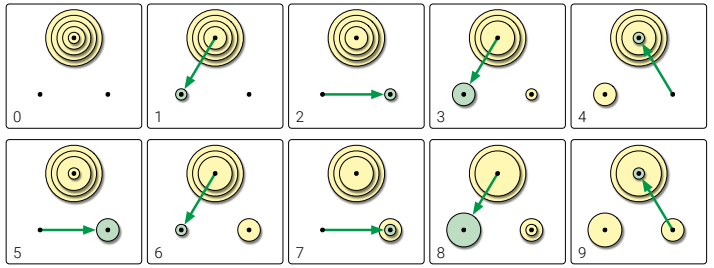


Figure 1.18. The first several moves in a counterclockwise Towers of Hanoi solution.

♣♥(c) Finally, suppose your only restriction is that you may never move a disk directly from peg 0 to peg 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? *[Hint: Matrices! This variant is considerably harder to analyze than the other two.]*

5. Consider the following more complex variant of the Tower of Hanoi puzzle The puzzle has a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

   (a) Describe a recursive algorithm for the case $k = 3$. *Exactly* how many moves does your algorithm make? (This is exactly the same as problem 4(a).)

   (b) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. *[Hint: Use part (a).]*

   ♥(c) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^2)$ moves. *[Hint: Don't use part (a).]*

   ♥(d) Describe a recursive algorithm for the case $k = \sqrt{n}$ that requires at most a polynomial number of moves. (Which polynomial??)

   ♥(e) Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n?

Recursion Trees

6. Use recursion trees to solve each of the following recurrences.

$$A(n) = 2A(n/4) + \sqrt{n} \quad B(n) = 2B(n/4) + n \quad C(n) = 2C(n/4) + n^2$$
$$D(n) = 3D(n/3) + \sqrt{n} \quad E(n) = 3E(n/3) + n \quad F(n) = 3F(n/3) + n^2$$
$$G(n) = 4G(n/2) + \sqrt{n} \quad H(n) = 4H(n/2) + n \quad I(n) = 4I(n/2) + n^2$$

7. Use recursion trees to solve each of the following recurrences.

   (j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

   (k) $K(n) = K(n/2) + 2K(n/3) + 3K(n/4) + n^2$

   (l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

♥8. Use recursion trees to solve each of the following recurrences.

   (m) $M(n) = 2M(n/2) + O(n \log n)$

   (n) $N(n) = 2N(n/2) + O(n/\log n)$

   (p) $P(n) = \sqrt{n}\,P(\sqrt{n}) + n$

   (q) $Q(n) = \sqrt{2n}\,Q(\sqrt{2n}) + \sqrt{n}$

Sorting

9. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

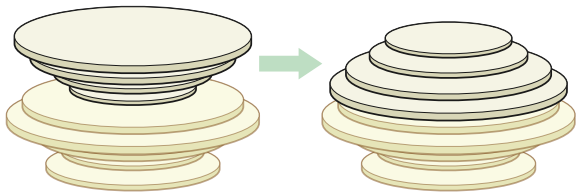


Figure 1.19. Flipping the top four pancakes.

   (a) Describe an algorithm to sort an arbitrary stack of n pancakes using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case?[15] *[Hint: This problem has nothing to do with the Tower of Hanoi.]*

[15]The *exact* worst-case optimal number of flips required to sort n pancakes (either burned or unburned) is an long-standing open problem; just do the best you can.

   (b) For every positive integer n, describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.

   (c) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case?

10. Recall that the *median-of-three* heuristic examines the first, last, and middle element of the array, and uses the median of those three elements as a quicksort pivot. Prove that quicksort with the median-of-three heuristic requires $\Omega(n^2)$ time to sort an array of size n in the worst case. Specifically, for any integer n, describe a permutation of the integers 1 through n, such that in every recursive call to median-of-three-quicksort, the pivot is always the second smallest element of the array. Designing this permutation requires intimate knowledge of the Partition subroutine.

   (a) As a warm-up exercise, assume that the Partition subroutine is *stable*, meaning it preserves the existing order of all elements smaller than the pivot, and it preserves the existing order of all elements smaller than the pivot.

   ♥(b) Assume that the Partition subroutine uses the specific algorithm listed on page 29, which is *not* stable.

11. (a) Hey, Moe! Hey, Larry! Prove that the following algorithm actually sorts its input!

```
StoogeSort(A[0..n-1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StoogeSort(A[0..m-1])
      StoogeSort(A[n-m..n-1])
      StoogeSort(A[0..m-1])
```

   (b) Would StoogeSort still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

   (c) State a recurrence (including the base case(s)) for the number of comparisons executed by StoogeSort.

   (d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

   (e) Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

12. The following cruel and unusual sorting algorithm was proposed by Gary Miller:

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                    ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                  ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i + n/4] ↔ A[i + n/2]
      UNUSUAL(A[1..n/2])                ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])              ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])           ⟨⟨recurse on middle half⟩⟩
```

The comparisons performed by this algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

13. An ***inversion*** in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

14. (a) Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. *[Hint: See the previous problem.]*

(b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

♥(c) Describe an algorithm for part (b) that runs in $O(n \log n)$ time. *[Hint: Use your solution from part (b)!]*

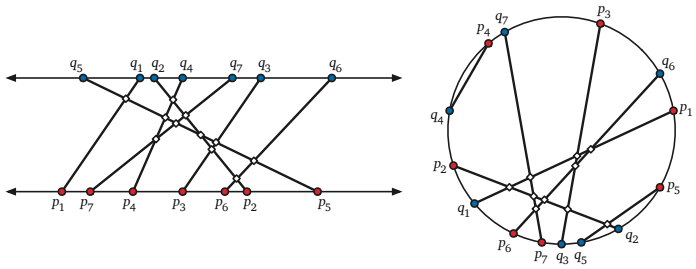


Figure 1.20. Eleven intersecting pairs of segments with endpoints on parallel lines, and ten intersecting pairs of segments with endpoints on a circle.

15. (a) Describe an algorithm that sorts an input array $A[1..n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A[k+1..k+\sqrt{n}]$ in place, given an arbitrary integer k between 0 and $n - \sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SQRTSORT; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?

♣(b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SQRTSORT, prove that no algorithm can sort using $o(f(n))$ calls to SQRTSORT.

(c) Now suppose SQRTSORT is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

Selection

16. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets

- $S_{<x}$ is the set of elements of S whose value is less than the value of x.
- $S_{>x}$ is the set of elements of S whose value is more than the value of x.

For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

17. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

    (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

    Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

18. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array.

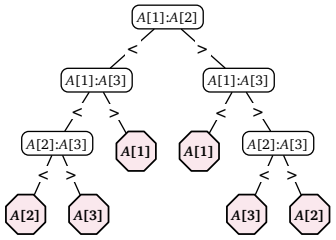


Figure 1.21. Finding the median of a 3-element array using at most 3 comparisons

19. Consider the generalization of the Blum-Floyd-Pratt-Rivest-Tarjan MOM-SELECT algorithm shown in Figure 1.22, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical.


```
MOM_bSELECT(A[1..n], k):
  if n ≤ b^2
    use brute force
  else
    m ← ⌈n/b⌉
    for i ← 1 to m
      M[i] ← MEDIANOFB(A[b(i−1)+1..bi])
    mom_b ← MOM_bSELECT(M[1..m], ⌊m/2⌋)
    r ← PARTITION(A[1..n], mom_b)
    if k < r
      return MOM_bSELECT(A[1..r−1], k)
    else if k > r
      return MOM_bSELECT(A[r+1..n], k−r)
    else
      return mom_b
```

Figure 1.22. A parametrized family of selection algorithms; see problem 19.

(a) State a recurrence for the running time of $\text{MOM}_b\text{SELECT}$, assuming that b is a constant (so the subroutine MEDIANOFB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately.

(b) What is the worst-case running time of $\text{MOM}_1\text{SELECT}$? *[Hint: This is a trick question.]*

♣♥(c) What is the worst-case running time of $\text{MOM}_2\text{SELECT}$? *[Hint: This is an unfair question!]*

♥(d) What is the worst-case running time of $\text{MOM}_3\text{SELECT}$? Finding an upper bound on the running time is straightforward; the hard part is showing that this analysis is actually tight. *[Hint: See problem 10.]*

♥(e) What is the worst-case running time of $\text{MOM}_4\text{SELECT}$? Again, the hard part is showing that the analysis cannot be improved.[16]

(f) For any constants $b \ge 5$, the algorithm $\text{MOM}_b\text{SELECT}$ runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \le 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

[16] The median of four elements is either the second smallest or the second largest. In 2014, Ke Chen and Adrian Dumitrescu proved that if we modify $\text{MOM}_4\text{SELECT}$ to find second-smallest elements when $k < n/2$ and second-largest elements when $k > n/2$, the resulting algorithm runs in $O(n)$ time! See their paper "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600) for details.

For each b between 5 and 13, find an upper bound on the running time of $\text{MOM}_b\text{SELECT}$ of the form $T(n) \le \alpha_b n$ for some explicit constant α_b. (For example, on page 39 we showed that $\alpha_5 \le 16$.)

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question!]*

20. Prove that the variant of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm shown in Figure 1.23, which uses an extra layer of small medians to choose the main pivot, runs in $O(n)$ time.

```
MOMOMSELECT(A[1..n], k):
  if n ≤ 81
      use brute force
  else
      m ← ⌈n/3⌉
      for i ← 1 to m
          M[i] ← MEDIANOF3(A[3i−2..3i])
      mm ← ⌈m/3⌉
      for j ← 1 to mm
          Mom[j] ← MEDIANOF3(M[3j−2..3j])
      momom ← MOMOMSELECT(Mom[1..mm], ⌊mm/2⌋)
      r ← PARTITION(A[1..n], momom)
      if k < r
          return MOMOMSELECT(A[1..r−1], k)
      else if k > r
          return MOMOMSELECT(A[r+1..n], k−r)
      else
          return momom
```

Figure 1.23. Selection by median of moms; see problem 20).

21. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(\log n)$ time. You can assume that the arrays contain no duplicate elements.

(b) Suppose we are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B$ in $\Theta(\log(m+n))$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$.) *[Hint: Use your solution to part (a).]*

♥(c) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

(d) Finally, suppose we are given a two dimensional array $A[1..m, 1..n]$ in which every row $A[i, \cdot]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Solve problem 16 first.]*

Arithmetic

22. In 1854, archaeologists discovered Sumerians clay tablets, carved around 2000BCE, that list the squares of integers up to 59. This discovery led some scholars to conjecture that ancient Sumerians performed multiplication by reduction to squaring, using an identity like $x \cdot y = (x^2 + y^2 - (x-y)^2)/2$. Unfortunately, those same scholars are silent on how the Sumerians supposedly squared larger numbers. Four thousand years later, we can finally rescue these Sumerian mathematicians from their lives of drudgery through the power of recursion!

(a) Describe a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\lg 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

(b) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

♥(c) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 5})$ time, by reducing to squaring only *five* $(n/3 + O(1))$-digit numbers. *[Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]*

23. (a) Describe and analyze a variant of Karatsuba's algorithm that multiplies any m-digit number and any n-digit number, for any $n \ge m$, in $O(nm^{\lg 3 - 1})$ time.

(b) Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time, using the algorithm from part (a) as a subroutine. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

(c) Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time. *[Hint: Watch out for an extra log factor in the running time.]*

♥(d) Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n) \log n)$ time. *[Hint: The analysis is the hard part; use a domain transformation.]*

24. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n:

```
Factorial(n):
  if n = 0
      return 1
  else
      return n · Factorial(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint:* $(n/2)^{n/2} < n! < n^n$*.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of Factorial. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using either the lattice algorithm or duplation and mediation. What is the running time of Factorial if we use this multiplication algorithm as a subroutine?

(d) The following recursive algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
Falling(n, m):          ⟨⟨Compute n!/(n − m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return Falling(n, ⌊m/2⌋) · Falling(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of $\text{Falling}(n, n)$ if we use grade-school multiplication? *[Hint: As usual, ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karatsuba's algorithm that multiplies any k-digit number and any l-digit number, for any $k \geq l$, in $O(k \cdot l^{\lg 3-1}) = O(k \cdot l^{0.585})$ time.

♥(f) What are the running times of $\text{Factorial}(n)$ and $\text{Falling}(n, n)$ if we use the modified Karatsuba multiplication from part (e)?

25. The ***greatest common divisor*** of two positive integer x and y, denoted $\mathbf{gcd}(x, y)$, is the largest integer d such that both x/d and y/d are integers. Euclid's *Elements*, written around 300BCE, describes the following recursive algorithm to compute $\gcd(x, y)$: [17]

[17] Euclid's algorithm is sometimes incorrectly described as the oldest recursive algorithm, or even the oldest *nontrivial* algorithm, even though the Egyptian duplation and mediation algorithm—which is both nontrivial and recursive—predates Euclid by at least 1500 years.


```
EuclidGCD(x, y):
  if x = y
      return x
  else if x > y
      return EuclidGCD(x − y, y)
  else
      return EuclidGCD(x, y − x)
```

(a) Prove that EuclidGCD correctly computes $\gcd(x, y)$.[18] Specifically:

i. Prove that $\text{EuclidGCD}(x, y)$ divides both x and y.

ii. Prove that every divisor of x and y is a divisor of $\text{EuclidGCD}(x, y)$.

(b) What is the worst-case running time of $\text{EuclidGCD}(x, y)$, as a function of x and y? (Assume that computing $x - y$ requires $O(\log x + \log y)$ time.)

(c) Prove that the following algorithm also computes $\gcd(x, y)$:

```
FastEuclidGCD(x, y):
  if y = 0
      return x
  else if x > y
      return FastEuclidGCD(y, x mod y)
  else
      return FastEuclidGCD(x, y mod x)
```

(d) What is the worst-case running time of $\text{FastEuclidGCD}(x, y)$, as a function of x and y? (Assume that computing x mod y takes $O(\log x \cdot \log y)$ time.)

(e) Prove that the following algorithm also computes $\gcd(x, y)$:

```
BinaryGCD(x, y):
  if x = y
      return x
  else if x and y are both even
      return 2 · BinaryGCD(x/2, y/2)
  else if x is even
      return BinaryGCD(x/2, y)
  else if y is even
      return BinaryGCD(x, y/2)
  else if x > y
      return BinaryGCD((x − y)/2, y)
  else
      return BinaryGCD(x, (y − x)/2)
```

[18] Euclid did not do this. Proposition 1 in *Elements* Book VII states that if $\text{EuclidGCD}(x, y) = 1$, then x and y are relatively prime (that is, $\gcd(x, y) = 1$), but the proof only considers the special case x mod $(y$ mod $(x$ mod $y)) = 1$. Proposition 2 states that if x and y are *not* relatively prime, then $\text{EuclidGCD}(x, y) = \gcd(x, y)$, but the proof only considers the special cases $\gcd(x, y) = y$ and $\gcd(x, y) = y$ mod $(x$ mod $y)$. Finally, these two Propositions do not make a complete proof that EuclidGCD is correct. Don't be like Euclid.

(f) What is the worst-case running time of BINARYGCD(x, y), as a function of x and y? (Assume that computing $x - y$ takes $O(\log x + \log y)$ time, and computing $z/2$ requires $O(\log z)$ time.)

Arrays

26. Suppose you are given a $2^n \times 2^n$ checkerboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by **L**-shaped tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time. *[Hint: First prove that such a tiling always exists.]*

27. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party by introducing them to each other. Members of the same political party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.[19]

    (a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

    (b) Now suppose there are more than two parties, but one party has a *plurality*: more people belong to that party than to any other party. Present a practical procedure to precisely pick the people from the plurality political party as parsimoniously as possible, presuming the plurality party is composed of at least p people. Pretty please.

28. Smullyan Island has three types of inhabitants: *knights* always speak the truth; *knaves* always lie; and *normals* sometimes speak the truth and sometimes don't. Everyone on the island knows everyone else's name and type (knight, knave, or normal). You want to learn the type of every inhabitant.

    You can ask any inhabitant to tell you the type of any other inhabitant. Specifically, if you ask "Hey X, what is Y's type?" then X will respond as follows:

[19] Real-world politics is much messier than this simplified model, but this is a theory book!

- If X is a knight, then X will respond with Y's correct type.
- If X is a knave, then X could respond with *either* of the types that Y is *not*.
- If X is a normal, then X could respond with *any* of the three types.

The inhabitants will ignore any questions not of this precise form; in particular, you may not ask an inhabitant about their own type. Asking the same inhabitant the same question multiple times always yields the same answer, so there's no point in asking any question more than once.

(a) Suppose you know that a strict majority of inhabitants are knights. Describe an efficient algorithm to identify the type of every inhabitant.

(b) Prove that if at most half the inhabitants are knights, it is impossible to determine the type of every inhabitant.

29. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

    Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.

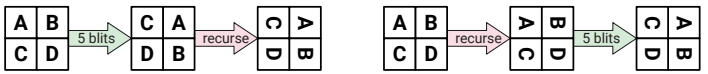


Figure 1.24. Two algorithms for rotating a pixel map.

(a) Prove that both versions of the algorithm are correct when n is a power of 2.

(b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

(d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(e) What if a $k \times k$ blit takes only $O(k)$ time?

30. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and

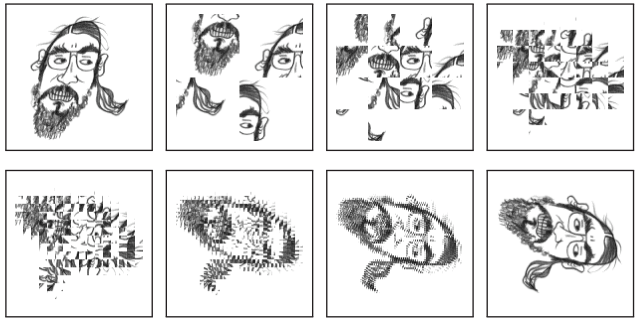


Figure 1.25. The first rotation algorithm (blit then recurse) in action. (See Image Credits at the end of the book.)

$A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3

is bitonic, but

3	6	9	8	7	5	1	2	4

is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

31. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

    (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

    (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

32. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

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

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

33. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that some prefix $A[1..k]$ is sorted in increasing order, the corresponding suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

    For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8

    (a) Describe and analyze an algorithm to compute the unknown integer k.

    (b) Describe and analyze an algorithm to determine if the given array contains a given number x.

34. At the end of the second act of the action blockbuster *Fast and Impossible XIII¾: The Last Guardians of Expendable Justice Reloaded*,the villainous Dr. Metaphor hypnotizes the entire Hero League/Force/Squad, arranges them in a long line at the edge of a cliff, and instructs each hero to shoot the closest taller heroes to their left and right, at a prearranged signal.

    Suppose we are given the heights of all n heroes, in order from left to right, in an array $Ht[1..n]$. (To avoid salary arguments, the producers insisted that no two heroes have the same height.) Then we can compute the Left and Right targets of each hero in $O(n^2)$ time using the following brute-force algorithm.

```
WhoTargetsWhom(Ht[1..n]):
  for j ← 1 to n
      ⟨⟨Find the left target L[j] for hero j⟩⟩
      L[j] ← None
      for i ← 1 to j−1
          if Ht[i] > Ht[j]
              L[j] ← i
      ⟨⟨Find the right target R[j] for hero j⟩⟩
      R[j] ← None
      for k ← n down to j+1
          if Ht[k] > Ht[j]
              R[j] ← k
  return L[1..n], R[1..n]
```

(a) Describe a divide-and-conquer algorithm that computes the output of WhoTargetsWhom in $O(n \log n)$ time.

(b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0..n-1]$ and $L[0..n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than None).

(c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose new targets, following the same algorithm, and then shoot their targets at the next signal. Eventually, only the shortest member of the Hero Crew/Alliance/Posse is left alive.[20]

Describe and analyze an algorithm to compute the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in $O(n)$ time.

35. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs $100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

(a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

♥(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

♣♥(c) Prove that your solution to part (b) is optimal up to a constant factor.

36. (a) Let $n = 2^\ell - 1$ for some positive integer ℓ. Suppose someone claims to hold an unsorted array $A[1..n]$ of *distinct* ℓ-bit strings; thus, exactly one ℓ-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function FetchBit(i, j), which returns the jth bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to FetchBit.

[20] In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society, saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures. So, yeah, basically it's *Avengers: Endgame*.

♥(b) Now suppose $n = 2^\ell - k$ for some positive integers k and ℓ, and again we are given an array $A[1..n]$ of *distinct* ℓ-bit strings. Describe an algorithm to find the k strings that are missing from A using only $O(n \log k)$ calls to FetchBit.

Trees

37. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree. See Figure 1.26 for an example.

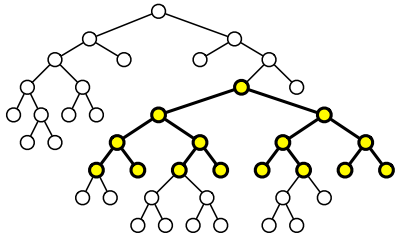


Figure 1.26. The largest complete subtree of this binary tree has depth 3.

38. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a ***central*** vertex if each of these smaller trees has at most $n/2$ vertices. See Figure 1.27 for an example.

Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree. *[Hint: First prove that every tree has a central vertex.]*

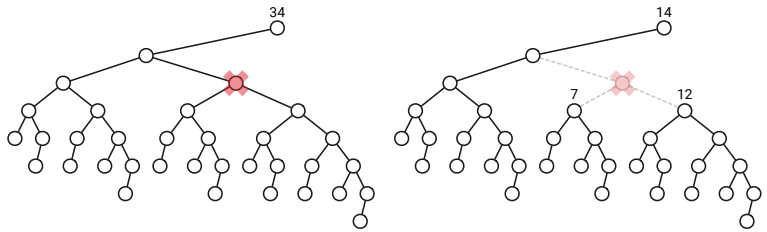


Figure 1.27. Deleting a central vertex in a 34-node binary tree, leaving subtrees with 14, 7, and 12 nodes.

39. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:
    - Preorder: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
    - Postorder: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

    Draw George's binary tree.

    (b) Recall that a binary tree is *full* if every non-leaf node has exactly two children.

    i. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

    ii. Prove that there is no algorithm to reconstruct an *arbitrary* binary tree from its preorder and postorder node sequences.

    (c) Describe and analyze a recursive algorithm to reconstruct an *arbitrary* binary tree, given its preorder and *inorder* node sequences as input.

    (d) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence.

    ♥(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence, ***in $O(n)$ time***.

    In parts (b)–(e), assume that all keys are distinct and that the input is consistent with at least one binary tree.

40. Suppose we have n points scattered inside a two-dimensional box. A ***kd-tree***[21] recursively subdivides the points as follows. If the box contains no points in its interior, we are done. Otherwise, we split the box into two smaller boxes with a *vertical* line, through a median point inside the box (*not* on its boundary), partitioning the points as evenly as possible. Then we recursively build a kd-tree for the points in each of the two smaller boxes, *after rotating them 90 degrees*. Thus, we alternate between splitting vertically and splitting horizontally at each level of recursion. The final empty boxes are called *cells*.

[21]The term "kd-tree" (pronounced "kay *dee* tree") was originally an abbreviation for "k-dimensional tree", but modern usage ignores this etymology, in part because nobody in their right mind would *ever* use the letter k to denote dimension instead of the *obviously* superior d. Etymological consistency would require calling the data structure in this problem a "2d-tree" (or perhaps a "2-d tree"), but the standard nomenclature is now "two-dimensional kd-tree". See also: B-tree (maybe), alpha shape, beta skeleton, epsilon net, Potomac River, Mississippi River, Lake Michigan, Lake Tahoe, Manhattan Island, La Brea Tar Pits, Sahara Desert, Mount Kilimanjaro, South Vietnam, East Timor, the Milky Way Galaxy, the City of Townsville, and self-driving automobiles.

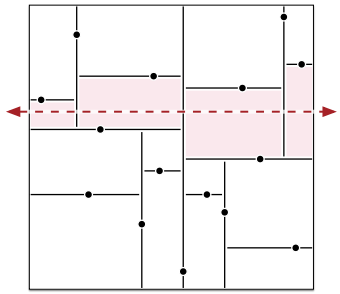


Figure 1.28. A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree containing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

♥41. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversals of binary trees. Bob sort-of understands the basic idea behind the traversal algorithms, but whenever he actually tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submits code with the following structure:

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    ■■Order(left(v))
    ■■Order(right(v))
```

```
InOrder(v):
  if v = Null
    return
  else
    ■■Order(left(v))
    print label(v)
    ■■Order(right(v))
```

```
PostOrder(v):
  if v = Null
    return
  else
    ■■Order(left(v))
    ■■Order(right(v))
    print label(v)
```

Each ■■ in this pseudocode hides one of the prefixes Pre, In, or Post. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

PreOrder($left(v)$)	PreOrder($right(v)$)
InOrder($left(v)$)	InOrder($right(v)$)
PostOrder($left(v)$)	PostOrder($right(v)$)

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

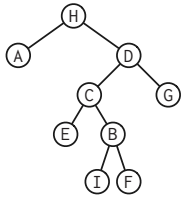
(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstructs Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

your first algorithm should return the following tree:



and your second algorithm should reconstruct the following code:

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    PreOrder(left(v))
    PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
    return
  else
    PostOrder(left(v))
    print label(v)
    PreOrder(right(v))
```

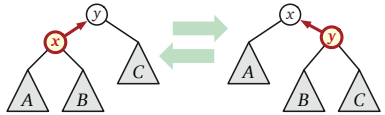
```
PostOrder(v):
  if v = Null
    return
  else
    InOrder(left(v))
    InOrder(right(v))
    print label(v)
```

♥42. Let T be a binary tree whose nodes store distinct numerical values. Recall that T is a ***binary search tree*** if and only if either (1) T is empty, or (2) T satisfies the following recursive conditions:

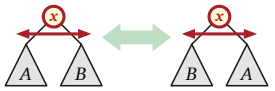
- The left subtree of T is a binary search tree.
- All values in the left subtree are smaller than the value at the root.
- The right subtree of T is a binary search tree.
- All values in the right subtree are larger than the value at the root.

Consider the following pair of operations on binary trees:

- ***Rotate*** an arbitrary node upward.[22]



- ***Swap*** the left and right subtrees of an arbitrary node.



In both of these operations, some, all, or none of the subtrees A, B, and C could be empty.

(a) Describe an algorithm to transform an *arbitrary* n-node binary tree with distinct node values into a binary search tree, using at most $O(n^2)$ rotations and swaps. Figure 1.29 shows a sequence of eight operations that transforms a five-node binary tree into a binary search tree.

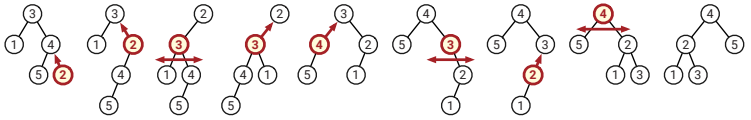


Figure 1.29. "Sorting" a binary tree: rotate 2, rotate 2, swap 3, rotate 3, rotate 4, swap 3, rotate 2, swap 4.

Your algorithm is not allowed to directly modify parent or child pointers, create new nodes, or delete old nodes; the *only* way to modify the tree is through rotations and swaps.

On the other hand, you may *compute* anything you like for free, as long as that computation does not modify the tree; the running time of your algorithm is *defined* to be the number of rotations and swaps that it performs.

♥(b) Describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree, using at most $O(n \log n)$ rotations and swaps.

[22] Rotations preserve the inorder sequence of nodes in a binary tree. Partly for this reason, rotations are used to maintain several types of balanced binary search trees, including AVL trees, red-black trees, splay trees, scapegoat trees, and treaps. See http://algorithms.wtf for lecture notes on most of these data structures.

(c) Prove that any n-node binary *search* tree can be transformed into any other binary *search* tree with the same node values, using only $O(n)$ rotations (and *no* swaps).

♥(d) **Open problem:** Either describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree using only $O(n)$ rotations and swaps, or prove that no such algorithm is possible. *[Hint: I don't think it's possible.]*

Where, however, the ambiguity cannot be cleared up, either by the rule of faith or by the context, there is nothing to hinder us to point the sentence according to any method we choose of those that suggest themselves.

— Augustine of Hippo, *De doctrina Christiana* (397CE)
Translated by Marcus Dods (1892)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin,
Scientific American (1886)

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— attributed to Bertrand Russell

When you come to a fork in the road, take it.

— Yogi Berra (giving directions to his house)

2

Backtracking

This chapter describes another important recursive strategy called ***backtracking***. A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates ***every*** alternative and then chooses the best one.

2.1 N Queens

The prototypical backtracking problem is the classical ***n Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens are attacking each other.

For readers not familiar with the rules of chess, this means that no two queens are in the same row, the same column, or the same diagonal.

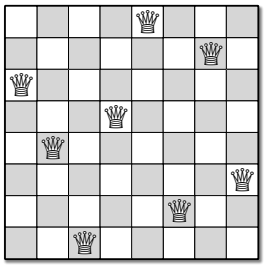


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array $[5, 7, 1, 4, 2, 8, 6, 3]$

In a letter written to his friend Heinrich Schumacher in 1850, the eminent mathematician Carl Friedrich Gauss wrote that one could easily confirm Franz Nauck's claim that the Eight Queens problem has 92 solutions by trial and error in a few hours. ("*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will.*") His description *Tatonniren* comes from the French *tâtonner*, meaning to feel, grope, or fumble around blindly, as if in the dark.

Gauss's letter described the following recursive strategy for solving the n-queens problem; the same strategy was described in 1882 by the French recreational mathematician Édouard Lucas, who attributed the method to Emmanuel Laquière. We place queens on the board one row at a time, starting with the top row. To place the rth queen, we methodically try all n squares in row r from left to right in a simple for loop. If a particular square is attacked by an earlier queen, we ignore that square; otherwise, we tentatively place a queen on that square and *recursively* grope for consistent placements of the queens in later rows.

Figure 2.2 shows the resulting algorithm, which recursively enumerates *all* complete n-queens solutions that are consistent with a given partial solution. Following Gauss, we represent the positions of the queens using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen. When PLACEQUEENS is called, the input parameter r is the index of the first empty row, and the prefix $Q[1..r-1]$ contains the positions of the first $r-1$ queens. In particular, to compute all n-queens solutions with no restrictions, we would call PLACEQUEENS$(Q[1..n], 1)$. The outer for-loop considers all possible placements of a queen on row r; the inner for-loop checks whether a candidate placement of row r is consistent with the queens that are already on the first $r-1$ rows.

The execution of PLACEQUEENS can be illustrated using a ***recursion tree***. Each node in this tree corresponds to a recursive subproblem, and thus to a legal partial solution; in particular, the root corresponds to the empty board