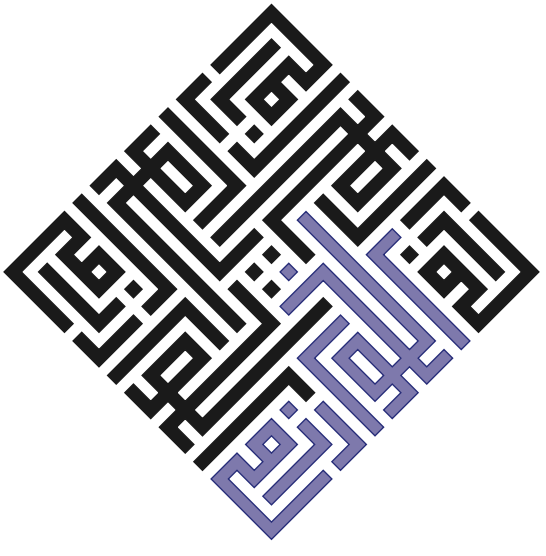
Algorithms



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!

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.

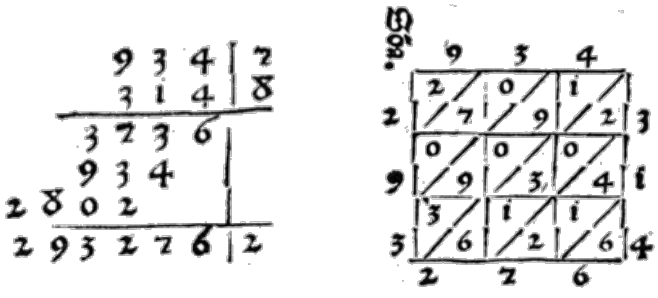


Figure 0.1. Computing 934 × 314 = 293276 using "long" multiplication (with error-checking by casting out nines) and "lattice" multiplication, from *L'Arte dell'Abbaco* (1458). (See Image Credits at the end of the book.)

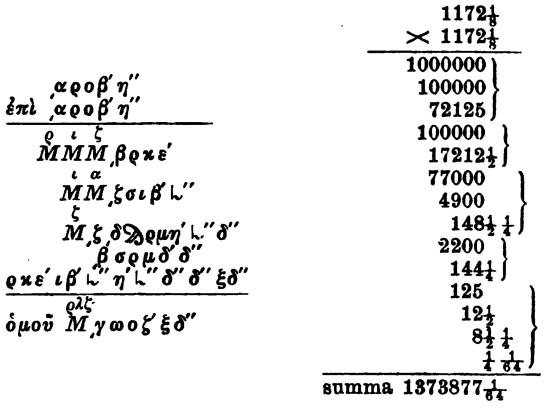


Figure 0.2. Eutocius's 6th-century calculation of $1172\frac{1}{8} \times 1172\frac{1}{8} = 1373877\frac{1}{64}$, in his commentary on Archimedes' *Measurement of the Circle*, transcribed (left) and translated into modern notation (right) by Johan Heiberg (1891). (See Image Credits at the end of the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

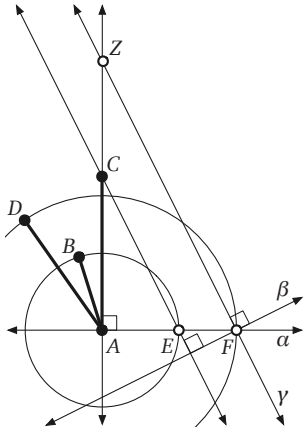


Figure 0.4. Multiplication by compass and straightedge.

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

[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.

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.

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.