

Algorithms from
THE BOOK

Copyright © 2020 by the Society for Industrial and Applied Mathematics

10 9 8 7 6 5 4 3 2 1


All rights reserved. Printed in the United States of America. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, write to the Society for Industrial and Applied Mathematics, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Julia is a trademark of JuliaLang.

This book is typeset using the AMS-developed LaTeX style files.

 Royalties from the sale of this book are placed in a fund to help students attend SIAM meetings and other SIAM-related activities. This fund is administered by SIAM, and qualified individuals are encouraged to write directly to SIAM for guidelines.

<i>Publications Director</i>	Kivmars H. Bowling
<i>Executive Editor</i>	Elizabeth Greenspan
<i>Developmental Editor</i>	Mellisa Pascale
<i>Managing Editor</i>	Kelly Thomas
<i>Production Editor</i>	David Riegelhaupt
<i>Copy Editor</i>	Claudine Dugan
<i>Production Manager</i>	Donna Witzleben
<i>Production Coordinator</i>	Cally A. Shrader
<i>Compositor</i>	Cheryl Hufnagle
<i>Graphic Designer</i>	Doug Smock

Library of Congress Cataloging-in-Publication Data

Names: Lange, Kenneth, author.

Title: Algorithms from THE BOOK / Kenneth Lange.

Description: Philadelphia : Society for Industrial and Applied Mathematics, [2020] | Includes bibliographical references and index.

Identifiers: LCCN 2019059040 (print) | LCCN 2019059041 (ebook) | ISBN 9781611976168 (paperback) | ISBN 9781611976175 (ebook)

Subjects: LCSH: Algorithms. | Computer algorithms. | Computer science--Mathematics.

Classification: LCC QA9.58 .L36 2020 (print) | LCC QA9.58 (ebook) | DDC 518.1--dc23

LC record available at <https://lcn.loc.gov/2019059040>

LC ebook record available at <https://lcn.loc.gov/2019059041>

siam is a registered trademark.

Contents

Preface	ix
<u>Chapter 1. Ancient Algorithms</u>	1
1.1. <u>Introduction</u>	1
1.2. <u>Peasant Multiplication</u>	1
1.3. <u>Babylonian Method</u>	2
1.4. <u>Quadratic Equations</u>	3
1.5. <u>Euclid's Algorithm</u>	4
1.6. <u>Sieve of Eratosthenes</u>	5
1.7. <u>Archimedes' Approximation of π</u>	5
1.8. <u>Problems</u>	8
<u>Chapter 2. Sorting</u>	11
2.1. <u>Introduction</u>	11
2.2. <u>Quicksort</u>	11
2.3. <u>Quickselect</u>	13
2.4. <u>Heapsort</u>	14
2.5. <u>Bisection</u>	16
2.6. <u>Priority Queues</u>	18
2.7. <u>Problems</u>	18
<u>Chapter 3. Graph Algorithms</u>	21
3.1. <u>Introduction</u>	21
3.2. <u>From Adjacency to Neighborhoods</u>	21
3.3. <u>Connected Components</u>	23
3.4. <u>Dijkstra's Algorithm</u>	24
3.5. <u>Prim's Algorithm</u>	26
3.6. <u>Problems</u>	28
<u>Chapter 4. Primality Testing</u>	29
4.1. <u>Introduction</u>	29
4.2. <u>Perfect Powers</u>	29
4.3. <u>Modular Arithmetic and Group Theory</u>	30
4.4. <u>Exponentiation in Modular Arithmetic</u>	31
4.5. <u>Fermat's Little Theorem</u>	31
4.6. <u>Miller–Rabin Test</u>	32
4.7. <u>Problems</u>	34
<u>Chapter 5. Solution of Linear Equations</u>	37
5.1. <u>Introduction</u>	37
5.2. <u>LU Decomposition and Gaussian Elimination</u>	37

5.3. Cholesky Decomposition	41
5.4. QR Decomposition and Gram–Schmidt Orthogonalization	44
5.5. Conjugate Gradient Method	47
5.6. Problems	50
Chapter 6. Newton’s Method	53
6.1. Introduction	53
6.2. Root Finding by Newton’s Method	53
6.3. Newton’s Method and Optimization	60
6.4. Variations on Newton’s Method	63
6.5. Problems	67
Chapter 7. Linear Programming	71
7.1. Introduction	71
7.2. Applications of Linear Programming	71
7.3. Revised Simplex Method	74
7.4. Revised Simplex Code	76
7.5. Kamarkar’s Algorithm	78
7.6. Problems	81
Chapter 8. Eigenvalues and Eigenvectors	85
8.1. Introduction	85
8.2. Applications of Eigen-decompositions	85
8.3. The Power Method and Markov Chains	86
8.4. Rayleigh Quotient Method	89
8.5. Householder Transformations	90
8.6. Divide and Conquer Spectral Decomposition	93
8.7. Jacobi’s Method	96
8.8. Extraction of the SVD	100
8.9. Problems	102
Chapter 9. MM Algorithms	105
9.1. Introduction	105
9.2. Majorization and Convexity	106
9.3. Sample MM Algorithms	108
9.4. Problems	116
Chapter 10. Data Mining	121
10.1. Introduction	121
10.2. k-Means Clustering	121
10.3. EM Clustering	123
10.4. Naive Bayes	126
10.5. k-Nearest Neighbors	128
10.6. Matrix Completion	129
10.7. Nonnegative Matrix Factorization	133
10.8. Problems	137
Chapter 11. The Fast Fourier Transform	141
11.1. Introduction	141
11.2. Basic Properties	141

11.3. Derivation of the Fast Fourier Transform	143
11.4. Approximation of Fourier Series Coefficients	145
11.5. Convolution	148
11.6. Fast Transforms and Matrix Factorization	152
11.7. Time Series	153
11.8. Problems	154
Chapter 12. Monte Carlo Methods	159
12.1. Introduction	159
12.2. Multiplicative Random Number Generators	159
12.3. Generation of Nonuniform Random Deviates	160
12.4. Randomized Matrix Multiplication	166
12.5. Markov Chain Monte Carlo	167
12.6. Simulated Annealing	173
12.7. Problems	174
Appendix A. Mathematical Review	179
A.1. Order Relations	179
A.2. Elementary Number Theory	180
A.3. Compactness in Mathematical Analysis	184
A.4. Convexity	185
A.5. Lagrange Multipliers	189
A.6. Linear Algebra	193
A.7. Banach's Contraction Mapping Theorem	201
Bibliography	203
Index	209

Preface

My inspiration for writing a survey of the best algorithms can be summarized by quoting Martin Aigner and Günter Ziegler, whose splendid book *Proofs from THE BOOK* is now in its fifth edition [2]. They write:

Paul Erdős liked to talk about THE BOOK, in which God maintains the perfect proofs for mathematical theorems, following the dictum of G. H. Hardy that there is no permanent place for ugly mathematics. Erdős also said that you need not believe in God but, as a mathematician, you should believe in THE BOOK.

Conversely, I would add that you need not believe in THE BOOK to believe in God. But I digress.

My more humble purpose is to highlight some of the most famous and successful algorithms and the lovely mathematics behind them. Algorithms are a dominant force in modern culture. Every time we turn on our browsers and commence a search, there stands an algorithm in the shadows. When we hop into our car, turn on the engine, and drive away, the motor and brakes obey hidden algorithms. Our banks and our spies depend on algorithms for encryption and decryption. The most important scientific instrument in any laboratory is the computer. Every indication is that algorithms will become more pervasive, not less. Thus, gaining an understanding of and a facility for designing algorithms is a worthy objective.

There is considerable debate about the top algorithms. The numerical analysts Don- garra and Sullivan [50] ignited the debate with their list:

- (1) Metropolis algorithm for Monte Carlo
- (2) Simplex method for linear programming
- (3) Krylov subspace iteration methods
- (4) The decompositional approach to matrix computations
- (5) The Fortran optimizing compiler
- (6) QR algorithm for computing eigenvalues
- (7) Quicksort algorithm for sorting
- (8) Fast Fourier transform
- (9) Integer relation detection
- (10) Fast multipole method

The applied mathematician Nicholas Higham updated this list in an influential blog post of March 29, 2016. His list shows six algorithms in common:

- (1) Newton and quasi-Newton methods
- (2) Matrix factorizations (LU, Cholesky, QR)
- (3) Singular value decomposition, QR and QZ algorithms
- (4) Monte-Carlo methods
- (5) Fast Fourier transform

- (6) Krylov subspace methods
- (7) JPEG
- (8) PageRank
- (9) Simplex algorithm
- (10) Kalman filter

In contrast, the computer scientist Marcos Otero in his blog entry of May 26, 2014, suggests

- (1) Merge Sort, Quick Sort, and Heap Sort
- (2) Fourier Transform and Fast Fourier Transform
- (3) Dijkstra's algorithm
- (4) RSA algorithm
- (5) Secure Hash Algorithm
- (6) Integer factorization
- (7) Link Analysis
- (8) Proportional Integral Derivative Algorithm
- (9) Data compression algorithms
- (10) Random Number Generation

Note the substantial divergence from the previous lists. In contrast, the data scientist James Le in his blog post of January 20, 2018, recommends

- (1) Linear regression
- (2) Logistic regression
- (3) Linear Discriminant Analysis
- (4) Classification and Regression Trees
- (5) Naive Bayes
- (6) K-Nearest Neighbors
- (7) Learning Vector Quantization
- (8) Support Vector Machines
- (9) Bagging and Random Forest
- (10) Boosting and AdaBoost

One lesson to be learned by this limited comparison is that there is no consensus. Our disciplinary backgrounds color our ranking of algorithms. Personally, I lean most toward Higham's list, with its heavy emphasis on linear algebra. However, it omits sorting and graph algorithms dear to computer scientists and regression and maximum likelihood algorithms dear to statisticians. The following pages adopt something from all four lists, as well as a few of my own favorites. Lacking usage statistics to back me up, my choices are personal and definitely subject to question.

Let me mention a few criteria guiding my exposition. I like a mathematical story. The mathematics need not be deep, but it should combine elements of surprise, ingenuity, and generality. Algorithms should be brief, easy to understand, and principled. Mere recipes without a defined objective hardly rise to the level of a legitimate algorithm. Readers may be offended by my glaring omissions. For example, I omit discussion of the QR algorithm for extracting the spectral decomposition of a symmetric matrix. I also emphasize sequential algorithms and barely mention parallel processing. Finally, except for the traveling salesman problem, I avoid NP hard problems altogether. This book represents my attempt to introduce students in the mathematical sciences to algorithms. Although it is celebratory, it is not intended as definitive or encyclopedic. In keeping with my desire to create a

textbook, each chapter ends with a problem section. Most problems are straightforward to solve, but a few might challenge even experts.

This brings us to the question of prerequisites. I assume readers have familiarity with linear algebra, advanced calculus, and probability. Prior exposure to numerical analysis would help but is not required. The appendices briefly sketch some theory pertinent to specific chapters. For instance, readers will definitely want to browse Appendix A.2 on elementary number theory before they tackle Chapter 4 on primality testing. Most chapters are isolated essays. A few, particularly Chapter 9 on data mining, rely on material from previous chapters. There is enough material here for a semester course at the pace of one chapter per week. UCLA, my home institution, operates on quarters, so I must pick and choose.

All of the algorithms discussed here are programmed in Julia and can be accessed at <https://bookstore.siam.org/ot168/bonus>. The advantage of actual code over flow charts is that students can readily experiment. Virtually all of my classroom students are proficient in higher-level languages such as R and MATLAB. Many are not adept with lower-level languages. Julia is a bridge language that combines coding simplicity with execution speed. I hope my code will be transparent to aficionados of C, Python, and other more popular languages. It is unfortunate that any language choice would disappoint many readers. Finally, let me stress that my code is apt to be much less efficient than production code. Julia programmers will almost certainly want to use Julia's base and library functions rather than my own.

For the record, here are some notation conventions used throughout the book. All vectors and matrices appear in boldface. The entries of the vectors $\mathbf{0}$ and $\mathbf{1}$ consist of 0's and 1's, respectively. The vector e_i has all entries 0 except a 1 in entry i . The 0/1 indicator of a set S is denoted by $1_S(\mathbf{x})$. The * superscript indicates a vector or matrix transpose. The Euclidean norm of a vector \mathbf{x} is denoted by $\|\mathbf{x}\|$ and the spectral and Frobenius norms of a matrix $\mathbf{M} = (m_{ij})$ by

$$\|\mathbf{M}\| = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{M}\mathbf{x}\|}{\|\mathbf{x}\|} \quad \text{and} \quad \|\mathbf{M}\|_F = \sqrt{\sum_i \sum_j m_{ij}^2},$$

respectively. All positive semidefinite matrices are symmetric by definition. When the difference $\mathbf{A} - \mathbf{B}$ of two symmetric matrices \mathbf{A} and \mathbf{B} is positive definite or positive semidefinite, we will write $\mathbf{A} \succ \mathbf{B}$ or $\mathbf{A} \succeq \mathbf{B}$. For a smooth real-valued function $f(\mathbf{x})$, we write its gradient (column vector of partial derivatives) as $\nabla f(\mathbf{x})$, its first differential (row vector of partial derivatives) as $df(\mathbf{x}) = \nabla f(\mathbf{x})^*$, and its second differential (Hessian matrix) as $d^2 f(\mathbf{x})$. If $g(\mathbf{x})$ is vector-valued with i th component $g_i(\mathbf{x})$, then the differential (Jacobi matrix) $dg(\mathbf{x})$ has i th row $dg_i(\mathbf{x})$.

I have many people to thank, not the least of all my wife, Genie, who graciously indulges my mathematical habits. Many UCLA colleagues, particularly, Jan de Leeuw, Robert Jennrich, Elliot Landaw, Stan Osher, Mary Sehl, Janet Sinsheimer, Eric Sobel, Marc Suchard, and Hua Zhou, have contributed to my growth as a scholar and a lover of algorithms. I conceived this book while on sabbatical at Stanford and owe a debt to the many inspiring statisticians there. The UCLA Biomathematics students Samuel Christensen, Ben Chu, Gabriel Hassler, Alfonso Landeros, and Tim Stutz contributed mightily to the fidelity and clarity of the text. Last of all, I would like thank my brothers, Eric, Fred, and John Lange, for their friendship, tolerance, and compassion. This book is dedicated to them.

CHAPTER 1

Ancient Algorithms

1.1. Introduction

We tend to think of ancient peoples as not quite as bright as we are. A more realistic view is that they simply lacked our enormous cultural inheritance in science, our complex technology, and our institutions of universal education. The examples covered in this chapter display some of the bursts of creativity by ancient mathematicians, physicists, astronomers, and philosophers. We are indebted to these past thinkers for constructing the first algorithms and setting forth principles that continue to guide algorithm development.

1.2. Peasant Multiplication

The history of the peasant multiplication algorithm is murky. Although it is often called the Russian peasant algorithm, the evidence suggests that it was known to the ancient Egyptians. The algorithm is recursive in nature and uses repeated doubling, halving, and addition. The basic idea is that to multiply two positive integers a and b , we can instead compute $(a/2) \cdot (2b)$ if a is even and $[(a-1)/2] \cdot (2b) + b$ if a is odd. In either case, a is reduced by at least a factor of 2, at the cost of one halving, one doubling, and possibly one addition. If we assume that halving, doubling, and addition are all constant-time operations, then the total computational complexity of the algorithm is proportional to the number of binary digits of a . In the following Julia code, the quantity $ab + c$ does not change from one pass to the next of the algorithm loop. Since the loop starts with the value ab , it also ends when $a = 1$ with this value. The algorithm is appealing because modern computers operate internally on binary numbers where doubling and halving reduce to bit shifting.

```
function peasantproduct(a::T, b::T) where T <: Integer
    c = zero(T)
    while a > one(T)
        if isodd(a)
            c = c + b
        end
        a = a >> 1 # divide a by 2
        b = b << 1 # multiply b by 2
    end
    return c + b
end
```

```
c = peasantproduct(10, 33)
```

It is worth emphasizing that peasant multiplication terminates after a finite number of steps with the correct answer. Other algorithms that we will meet later share this property, provided we entertain the fiction that computers are capable of exact arithmetic with real

numbers. The next algorithm is representative of the class of iterative algorithms. These converge over an infinite number of steps to a correct answer. The rate of convergence of an iterative algorithm is critically important in evaluating its performance. This must be balanced against the computational complexity of each step.

1.3. Babylonian Method

The ancient Babylonians discovered a lovely algorithm for extracting the square root of a nonnegative number c . Their algorithm turns out to be a special case of Newton's method. The Babylonian iterations are defined by the formula

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{c}{x_n} \right)$$

with a positive initial value x_0 . The choice $x_0 = 1$ is somewhat neutral but hardly optimal on a computer with binary arithmetic. The Babylonian update can be motivated by solving the approximation

$$c = (x_n + \Delta)^2 = x_n^2 + 2x_n\Delta + \Delta^2 \approx x_n^2 + 2x_n\Delta$$

for Δ and then calculating

$$x_n + \Delta \approx x_n + \frac{1}{2x_n}(c - x_n^2).$$

Let us show that regardless of the choice of x_0 , the iterates converge to \sqrt{c} . Consider the difference

$$\begin{aligned} (1.1) \quad x_{n+1} - \sqrt{c} &= \frac{1}{2} \left(x_n + \frac{c}{x_n} \right) - \sqrt{c} \\ &= \frac{x_n^2 + c - 2\sqrt{c}x_n}{2x_n} \\ &= \frac{(x_n - \sqrt{c})^2}{2x_n}. \end{aligned}$$

This representation makes it clear that $x_{n+1} \geq \sqrt{c}$ regardless of the value of x_n . Furthermore, $x_{n+1} = \sqrt{c}$ if and only if $x_n = \sqrt{c}$.

If we assume that $x_n > \sqrt{c}$, then the difference formula (1.1) implies that

$$x_{n+1} - \sqrt{c} < x_n - \sqrt{c}$$

if and only if $x_n - \sqrt{c} < 2x_n$, a condition which is obvious. Thus, the iterates decrease and possess a limit x_∞ . This limit satisfies the equation

$$x_\infty = \frac{1}{2} \left(x_\infty + \frac{c}{x_\infty} \right),$$

whose only solution is \sqrt{c} . The rate of convergence of x_n to \sqrt{c} is quadratic because for large n

$$x_{n+1} - \sqrt{c} \approx \frac{(x_n - \sqrt{c})^2}{2\sqrt{c}}.$$

In practice the number of significant digits in x_n doubles at each iteration.

TABLE 1.1. The Babylonian Method Applied to π^2

Iteration n	x_n	$x_n - \pi$
0	1.0	-2.141592653589793
1	5.434802200544679	2.293209546954886
2	3.625401431921964	0.483808778332171
3	3.173874724746142	0.032282071156349
4	3.141756827069927	0.000164173480134
5	3.141592657879262	$4.289468336 \times 10^{-9}$
6	3.141592653589793	0.0

Here is Julia code implementing the Babylonian method:

```
function babylonian(c::T, tol::T) where T <: Real
    x = one(T) # start x at 1
    while abs(x^2 - c) > tol # convergence test
        x = (x + c / x) / 2
    end
    return x
end
```

```
root = babylonian(pi^2, 1e-10)
```

Table 1.1 records the Babylonian method applied to π^2 . Six iterations suffice to achieve convergence to machine precision. Once the iterates reach the vicinity of π , the fast quadratic rate of convergence kicks in.

1.4. Quadratic Equations

Most educated people are at least vaguely familiar with the solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

to the quadratic equation $ax^2 + bx + c = 0$. These two roots are an immediate consequence of the identity

$$\left(x + \frac{b}{2a}\right)^2 = \frac{b^2 - 4ac}{4a^2}.$$

The ancient civilizations of Babylonia, Egypt, Greece, China, and India all made contributions in solving quadratics. The quadratic formula in the form we know today is credited to René Descartes. Let us stress that the quadratic formula is a legitimate algorithm even though it is not explicitly iterative. Implicitly it is iterative because extracting square roots is usually iterative.

Here we would like to comment on the potential loss of numerical accuracy in applying the classical quadratic formula. Roundoff error occurs when two numbers of the same sign and approximately the same magnitude are subtracted. Assuming the two roots are real, the smaller root in magnitude is the one liable to catastrophic cancellation. To avoid roundoff, one should compute the larger root r in magnitude and exploit the fact that the product of the two roots equals $\frac{c}{a}$. The following Julia code does exactly this. Note that roundoff is not an issue when the roots are complex. This case is handled separately in the Julia code.

```

function quadratic(a::T, b::T, c::T) where T <: Real
    d = b^2 - 4a * c # discriminant
    if d > zero(T)
        if b >= zero(T)
            r1 = (-b - sqrt(d)) / (2a)
        else
            r1 = (-b + sqrt(d)) / (2a)
        end
        r2 = c / (r1 * a)
        return (r1, r2)
    else
        return (-b + sqrt(d + 0im)) / (2a), (-b - sqrt(d + 0im)) / (2a)
    end
end

```

```

(a, b, c) = (1.0, -2.0, 1.0)
(r1, r2) = quadratic(a, b, c)

```

For the sake of simplicity, this function ignores the admonition of many Julia experts that returned values be type stable.

1.5. Euclid's Algorithm

Euclid's algorithm is an efficient method for computing the greatest common divisor (gcd) of two integers $a > b > 0$. By definition $\gcd(a, b)$ is the largest integer that divides both a and b without leaving a remainder. The algorithm appears in *Euclid's Elements* (circa 300 BC). It can be used to reduce fractions to their simplest forms and occurs in many number-theoretic and cryptographic calculations.

Suppose a and b have greatest common divisor c . As Proposition A.2.1 of Appendix A.2 demonstrates, there exists a unique pair of integers q and r such that $a = qb + r$ and $0 \leq r < b$. If by chance $r = 0$, then clearly $\gcd(a, b) = b$. Otherwise, note that since c divides both a and b , it must divide r as well. Conversely, any integer d dividing both b and r must divide a . It follows that $\gcd(a, b) = \gcd(b, r)$, and we can replace a by b and b by r . Because $b < a$ and $r < b$, this replacement process must come to an end in a finite number of steps with $a = b$ or $a > b$ and $r = 0$. At that point we can read off the greatest common divisor $\gcd(a, b) = b$.

```

function euclid(m::T, n::T) where T <: Integer
    (a, b) = (m, n)
    while b != zero(T)
        (a, b) = (b, rem(a, b))
    end
    return a
end

```

```
gcd = euclid(600, 220)
```

There is a matrix form of Euclid's algorithm. Note that

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} b \\ r \end{pmatrix} = M \begin{pmatrix} b \\ r \end{pmatrix}$$

and that

$$M^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$$

exists and possesses integer entries. If Euclid's algorithm takes s steps and ends with $[\gcd(a, b), 0]$, then

$$\begin{pmatrix} a \\ b \end{pmatrix} = M_1 \cdots M_s \begin{pmatrix} \gcd(a, b) \\ 0 \end{pmatrix}.$$

It follows that

$$\begin{pmatrix} \gcd(a, b) \\ 0 \end{pmatrix} = M_s^{-1} \cdots M_1^{-1} \begin{pmatrix} a \\ b \end{pmatrix}$$

and in particular that

$$(1.2) \quad \gcd(a, b) = ca + db$$

for integers c and d . This result was first proved by Bézout.

1.6. Sieve of Eratosthenes

Eratosthenes' algorithm finds all primes between 1 and a fixed integer n . Eratosthenes of Cyrene (circa 276 BC to 195/194 BC) was a Greek mathematician and astronomer. He served as director of the famous Library of Alexandria and is best known for calculating the circumference of the earth. His sieve works by marking as composite (non-prime) all of the multiples of each prime, starting with all multiples of 2, then all multiples of 3, and so forth. The multiples of a given prime p are generated in the order $p, 2p, 3p, \dots$ until n is reached. The sieve avoids testing each candidate number for divisibility by each prime. It also relies on the simple fact that when a number is composite, one of its factors is less than or equal to \sqrt{n} . Here is Julia code implementing the sieve.

```
function eratosthenes(n::Integer)
    isprime = trues(n)
    isprime[1] = false # 1 is composite
    for i = 2:round(Int, sqrt(n))
        if isprime[i]
            for j = i^2:i:n # all multiples of i < i^2 already composite
                isprime[j] = false
            end
        end
    end
    return filter(x -> isprime[x], 1:n) # eliminate composite numbers
end
```

```
prime_list = eratosthenes(100)
```

1.7. Archimedes' Approximation of π

Around 250 BC, the Greek mathematician Archimedes derived an algorithm for approximating π , the ratio of a circle's circumference to its diameter. He was able to show that $3\frac{10}{71} < \pi < 3\frac{1}{7}$ by considering the length b_n of the perimeter of a regular polygon with $3 \cdot 2^n$ sides inscribed within a circle and the length a_n of the perimeter of a regular polygon with $3 \cdot 2^n$ sides circumscribed outside a circle. For a circle with diameter 1, $b_n < \pi < a_n$.

Starting with the known values of a_1 and b_1 for circumscribing and inscribing hexagons, he was able to construct a recurrence relation connecting a_{n+1} and b_{n+1} to a_n and b_n . By doubling the number of sides of the initial hexagons to 12-sided polygons, then to 24-sided polygons, and ultimately to 96-sided polygons, Archimedes was able to bring the two perimeters ever closer in length to the circumference of the circle. He, like all Greek mathematicians, relied heavily on geometric arguments.

Figure 1.1 depicts a circle with diameter 1 and corresponding inscribed and circumscribed squares. Squares are simpler initial figures than hexagons. The perimeter lengths of the two squares are $b_0 = 4/\sqrt{2} = 2\sqrt{2}$ and $a_0 = 4$. We now explore how Archimedes calculated perimeter lengths for inscribed and circumscribed polygons after doubling the number of sides. Call the perimeter lengths b_n and a_n for regular polygons with $2 \cdot 2^n$ sides. Archimedes' argument depends on four facts: (a) at each point of a circle the tangent line and the radial line from the center are perpendicular, (b) two regular polygons of the same number of sides share interior angles, (c) two right triangles are similar if they share a minor angle, and (d) side lengths in similar triangles occur in a constant ratio. Thus, triangles HDG and AFH are similar, as are triangles ACD and AFH and the isosceles triangles DFB and DGF.

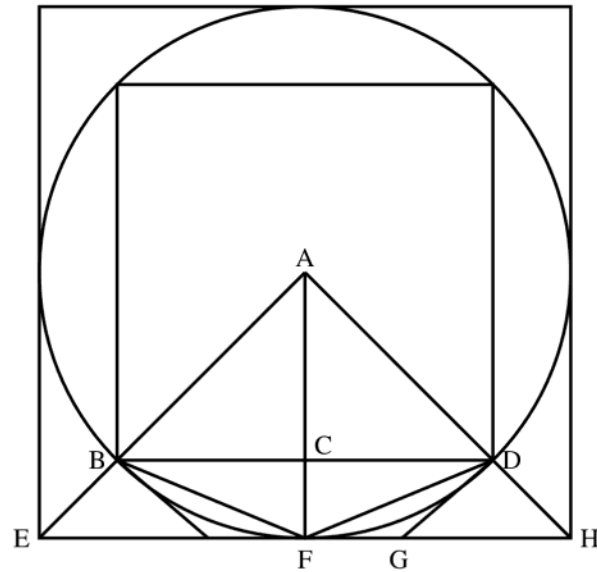


FIGURE 1.1. Archimedes' Polygons for Estimating π

If we let $m = 2 \cdot 2^n$ be the number of sides of the two regular polygons, then the similarity of triangles HDG and AFH implies that

$$\frac{\frac{a_{n+1}}{4m}}{\frac{a_n}{2m} - \frac{a_{n+1}}{4m}} = \frac{DG}{GH} = \frac{AF}{AH}.$$

Likewise, the similarity of triangles ACD and AFH implies that

$$\frac{\frac{b_n}{2m}}{\frac{a_n}{2m}} = \frac{CD}{AF} = \frac{AD}{AH}.$$

Since $AF = AD = \frac{1}{2}$ is the radius of the circle,

$$\frac{\frac{a_{n+1}}{4m}}{\frac{a_n}{2m} - \frac{a_{n+1}}{4m}} = \frac{\frac{b_n}{2m}}{\frac{a_n}{2m}}.$$

This equation is equivalent to the equation

$$(1.3) \quad a_{n+1} = \frac{2a_n b_n}{a_n + b_n}.$$

Finally, the similarity of the isosceles triangles DFB and DGF yields

$$\frac{\frac{b_n}{m}}{\frac{b_{n+1}}{2m}} = \frac{\frac{b_{n+1}}{2m}}{\frac{a_{n+1}}{4m}},$$

which is equivalent to

$$(1.4) \quad b_{n+1} = \sqrt{a_{n+1} b_n}.$$

The two equations (1.3) and (1.4) constitute Archimedes' recurrence scheme for approximating π .

Here is Julia code implementing Archimedes' algorithm.

```
function archimedes(tol::T) where T <: Real
    (a, b) = (4 * one(T), 2 * sqrt(2 * one(T)))
    while abs(a - b) > tol
        a = 2 * a * b / (a + b)
        b = sqrt(a * b)
    end
    return (a, b)
end
```

```
(upper, lower) = archimedes(1e-6)
```

Archimedes lacked most of the tools of calculus. Let us now show how these tools provide insight into the rate of convergence of the sequences a_n and b_n to π . If we assume by induction that $b_n < a_n$, then

$$\begin{aligned} a_{n+1} &= \frac{2a_n b_n}{a_n + b_n} < \frac{2a_n b_n}{2b_n} = a_n \\ a_{n+1} &= \frac{2a_n b_n}{a_n + b_n} > \frac{2a_n b_n}{2a_n} = b_n. \end{aligned}$$

These two inequalities in turn imply that

$$\begin{aligned} b_{n+1} &= \sqrt{a_{n+1} b_n} < a_{n+1} \\ b_{n+1} &= \sqrt{a_{n+1} b_n} > b_n. \end{aligned}$$

TABLE 1.2. Archimedes' Approximation Scheme for π

Iteration n	a	b
0	4.0	2.8284271
1	3.3137085	3.0614675
2	3.1825979	3.1214452
3	3.1517249	3.1365485
4	3.1441184	3.1403312
5	3.1422236	3.1412773
6	3.1417504	3.1415138
7	3.1416321	3.1415729
8	3.1416025	3.1415877
9	3.1415951	3.1415914
10	3.1415933	3.1415923

Thus, as n tends to ∞ , the monotone sequence a_n decreases to a limit a , the monotone sequence b_n increases to a limit b , and $b \leq a$. The gap $a_n - b_n$ satisfies

$$\begin{aligned}
 a_{n+1} - b_{n+1} &\leq a_{n+1} - b_n \\
 &= \frac{2a_n b_n}{a_n + b_n} - b_n \\
 &= \frac{b_n}{a_n + b_n} (a_n - b_n) \\
 &\leq \frac{1}{2} (a_n - b_n).
 \end{aligned}$$

Thus, the gap is more than halved at each iteration. It follows that $a = b$ and that each iteration yields roughly another binary digit of accuracy. Table 1.2 displays the linear rate of convergence of Archimedes' algorithm.

1.8. Problems

- (1) The Goldschmidt method of division reduces the evaluation of a fraction $\frac{a}{b}$ to addition and multiplication. By bit shifting we may assume that $b \in (\frac{1}{2}, 1]$. Replace b by $1 - x$ and write

$$\begin{aligned}
 \frac{a}{1-x} &= \frac{a(1+x)}{1-x^2} \\
 &= \frac{a(1+x)(1+x^2)}{1-x^4} \\
 &= \frac{a(1+x)(1+x^2)\cdots(1+x^{2^{n-1}})}{1-x^{2^n}}.
 \end{aligned}$$

Program this algorithm in Julia. How large should n be so that the denominator is effectively 1? Note that the powers of x should be computed by repeated squaring.

- (2) Use the significant and exponent functions of Julia and devise a better initial value than $x_0 = 1.0$ for the Babylonian method. Explain your choice, and test it on a few examples.

- (3) For
- $c \geq 0$
- show that the iteration scheme

$$x_{n+1} = \frac{c + x_n}{1 + x_n}$$

converges to \sqrt{c} starting from any $x_0 \geq 0$. Verify either theoretically or empirically that the rate of convergence is much slower than that of the Babylonian method.

- (4) Dedekind's algorithm for extracting
- \sqrt{c}
- iterates according to

$$x_{n+1} = \frac{x_n(x_n^2 + 3c)}{3x_n^2 + c}.$$

Program Dedekind's algorithm in Julia. Demonstrate cubic convergence by deriving the identity

$$x_{n+1}^2 - c = \frac{(x_n^2 - c)^3}{(3x_n^2 + c)^2}.$$

Finally, argue that Dedekind's algorithm converges to \sqrt{c} regardless of the initial value $x_0 > 0$.

- (5) Find coefficients (a, b, c) where the standard quadratic formula is grossly inaccurate when implemented in single precision. You will have to look up how to represent single precision numbers in Julia.
- (6) Why does the product of the two roots of a quadratic equal $\frac{c}{a}$?
- (7) Solving a cubic equation $ax^3 + bx^2 + cx + d = 0$ is much more complicated than solving a quadratic. Demonstrate that (a) the substitution $x = y - \frac{b}{3a}$ reduces the cubic to $y^3 + ey + f = 0$ for certain coefficients e and f , (b) the further substitution $y = z - \frac{e}{3z}$ reduces this equation to $z^6 + fz^3 - \frac{e^3}{27}$, and (c) the final substitution $w = z^3$ reduces the equation in z to a quadratic in w , which can be explicitly solved. One can now reverse these substitutions and capture six roots, which collapse in pairs to at most three unique roots. Program your algorithm in Julia, and make sure that it captures complex as well as real roots.
- (8) Write a Julia program to find the integers c and d in Bézout's identity

$$\gcd(a, b) = ca + db.$$

- (9) The prime number theorem says that the number of primes $\pi(n)$ between 1 and n is asymptotic to $\frac{n}{\ln n}$. Use the Sieve of Eratosthenes to check how quickly the ratio $\frac{\pi(n) \ln(n)}{n}$ tends to 1.
- (10) A Pythagorean triple (a, b, c) satisfies $a^2 + b^2 = c^2$. Given an array \mathbf{x} of positive integers, write a Julia program to find all Pythagorean triples in \mathbf{x} . (Hint: Replace the entries of \mathbf{x} by their squares and sort the result.)
- (11) Show that the perimeter lengths a_n and b_n in Archimedes' algorithm satisfy

$$a_n = m \tan \frac{\pi}{m} \quad \text{and} \quad b_n = m \sin \frac{\pi}{m},$$

where $m = 2 \cdot 2^n$ is the number of sides of the two regular polygons. Use this representation and appropriate trigonometric identities to prove the recurrence relations (1.3) and (1.4).

- (12) Based on the trigonometric representations of the previous problem, show that $\frac{1}{3}a_n + \frac{2}{3}b_n$ is a much better approximation to π than either a_n or b_n [142]. Check your theoretical conclusions by writing a Julia program that tracks all three approximations to π .

- (13) Consider evaluation of the polynomial

$$p(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$$

for a given value of x . If one proceeds naively, then it takes $n - 1$ multiplications to form the powers $x^k = x \cdot x^{k-1}$ for $2 \leq k \leq n$, n multiplications to multiply each power x^k by its coefficient a_{n-k} , and n additions to sum the resulting terms. This amounts to $3n - 1$ operations in all. A more efficient method exploits the fact that $p(x)$ can be expressed as

$$\begin{aligned} p(x) &= x(a_0x^{n-1} + a_1x^{n-2} + \cdots + a_{n-1}) + a_n \\ &= xb_{n-1}(x) + a_n. \end{aligned}$$

Since the polynomial $b_{n-1}(x)$ of degree $n - 1$ can be similarly reduced, a complete recursive scheme for evaluating $p(x)$ is given by

$$b_0(x) = a_0, \quad b_k(x) = xb_{k-1}(x) + a_k, \quad k = 1, \dots, n.$$

This scheme requires only n multiplications and n additions in order to compute $p(x) = b_n(x)$. Program the scheme and extend it to the simultaneous evaluation of the derivative $p'(x)$ of $p(x)$.

- (14) Consider a sequence
- x_1, \dots, x_n
- of
- n
- real numbers. After you have computed the sample mean and variance

$$\mu_n = \frac{1}{n} \sum_{i=1}^n x_i \quad \text{and} \quad \sigma_n^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_n)^2,$$

suppose you are presented with a new observation x_{n+1} . It is possible to adjust the sample mean and variance without revisiting all of the previous observations. Verify theoretically and then code the updates

$$\begin{aligned} \mu_{n+1} &= \frac{1}{n+1} (n\mu_n + x_{n+1}) \\ \sigma_{n+1}^2 &= \frac{n}{n+1} \sigma_n^2 + \frac{1}{n} (x_{n+1} - \mu_{n+1})^2. \end{aligned}$$

CHAPTER 2

Sorting

2.1. Introduction

Sorting lists of items such as numbers or words is one of the most thoroughly studied tasks in computer science [100]. The number of sorting algorithms is large and growing. We will focus on two of the most popular, quicksort and heapsort. Sorting algorithms can be compared in at least five different ways [171]: (a) average computational complexity, (b) worst case computational complexity, (c) required computer storage, (d) ability to take advantage of existing order, and (e) ease and clarity of coding. Both quicksort and heapsort sort an array in place. Both enjoy an average computational complexity of $O(n \ln n)$ for n items. Quicksort tends to be quicker in practice. Quicksort has worst case computational complexity $O(n^2)$, while heapsort retains its $O(n \ln n)$ complexity. Both fail miserably in recognizing existing order. Finally, both are straightforward to code.

2.2. Quicksort

Quicksort [85] is possibly the most elegant sorting algorithm. It operates by a divide and conquer principle that picks a pivot entry of the underlying list and partitions the list around the picked pivot. For the sake of simplicity, we will assume the list is a sequence of n numbers. In the version of quicksort explained here, the choice of the pivot entry is random. Randomness at this level plus the randomness of the sequence itself facilitate calculation of the average number a_n of operations (comparisons and swaps) required to sort the sequence. A recurrence relation for a_n lies at the heart of this calculation. Solution of the recurrence relation shows that $a_n = 2n \ln n$ to a good approximation.

Hoare's quicksort algorithm is based on the idea of finding a splitting entry x_i of a sequence x_1, \dots, x_n of n distinct numbers in the sense that $x_j < x_i$ for $j < i$ and $x_j > x_i$ for $j > i$. In other words, a splitter x_i is already correctly ordered relative to the rest of the entries of the sequence. Finding a splitter reduces the computational complexity of sorting because it is easier to sort both of the subsequences x_1, \dots, x_{i-1} and x_{i+1}, \dots, x_n than it is to sort the original sequence. At this juncture, one can reasonably object that no splitter need exist, and even if one does, it may be difficult to locate. The quicksort algorithm avoids these difficulties by randomly selecting a splitting value and then slightly rearranging the sequence so that this splitting value occupies the correct splitting location.

In the background of quicksort is the probabilistic assumption that all $n!$ permutations of the n values are equally likely. The algorithm begins by randomly selecting one of the n values and moving it to the leftmost or first position of the sequence. Through a sequence of exchanges, this value is then promoted to its correct location. In the probabilistic setting adopted, the correct location of the splitter is uniformly distributed over the n positions of the sequence.

The promotion process works by exchanging or swapping entries to the right of the randomly chosen splitter x_1 , which is kept in position 1 until a final swap. Let j be our

current position in the sequence as we examine it from left to right. In the sequence up to position j , a candidate position i for the insertion of x_1 must satisfy the conditions $x_k \leq x_1$ for $1 < k \leq i$ and $x_k > x_1$ for $i < k \leq j$. At position $j = 1$, we are forced to put $i = 1$. Now suppose we have successfully advanced to a general position j and identified a corresponding candidate position i . To move from position j to position $j + 1$, we examine x_{j+1} . If $x_{j+1} > x_1$, then we keep the current candidate position i . On the other hand, if $x_{j+1} \leq x_1$, then we swap x_{i+1} and x_{j+1} and replace i by $i + 1$. In either case, the two required conditions imposed on i continue to hold in moving from position j to position $j + 1$. It is now clear that we can inductively march from the left end to the right end of the sequence, carrying out a few swaps in the process, so that when $j = n$, the value i marks the correct position to insert x_1 . Once this insertion is made, the subsequences x_1, \dots, x_{i-1} and x_{i+1}, \dots, x_n can be sorted separately by the same splitting procedure.

The following recursive Julia code implements quicksort on any sortable list of items. We illustrate the algorithm on integers and letters.

```
function quicksort(x::Vector, left = 1, right = length(x))
    i = rand(left:right) # select a random splitting value
    split = x[i]
    (x[left], x[i]) = (split, x[left])
    i = left
    for j = (left + 1):right # position the splitting value
        if x[j] <= split
            i = i + 1
            (x[i], x[j]) = (x[j], x[i])
        end
    end
    (x[left], x[i]) = (x[i], split)
    if i > left + 1 # sort to the left of the value
        quicksort(x, left, i - 1)
    end
    if i + 1 < right # sort to the right of the value
        quicksort(x, i + 1, right)
    end
end

x = [5, 4, 3, 1, 2, 8, 7, 6, -1];
quicksort(x)
println(x)
x = ['a', 'c', 'd', 'b', 'f', 'e', 'h', 'g', 'y'];
quicksort(x)
println(x)
```

To explore the average behavior of quicksort, let a_n be the expected number of operations involved in quick sorting a sequence of n numbers. By convention $a_0 = 0$. If we base our analysis only on how many positions j must be examined at each stage and not on how many swaps are involved, then we can write the recurrence relation

$$(2.1) \quad a_n = n - 1 + \frac{1}{n} \sum_{i=1}^n (a_{i-1} + a_{n-i}) = n - 1 + \frac{2}{n} \sum_{i=1}^n a_{i-1}$$

by conditioning on the correct position i of the first splitter.

The recurrence relation (2.1) looks formidable, but a few algebraic maneuvers render it solvable. Multiplying equation (2.1) by n produces

$$na_n = n(n-1) + 2 \sum_{i=1}^n a_{i-1}.$$

If we subtract from this the corresponding expression for $(n-1)a_{n-1}$, then we get

$$na_n - (n-1)a_{n-1} = 2n - 2 + 2a_{n-1},$$

which can be rearranged to give

$$(2.2) \quad \frac{a_n}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{a_{n-1}}{n}.$$

Equation (2.2) can be iterated to yield

$$\begin{aligned} \frac{a_n}{n+1} &= 2 \sum_{k=1}^n \frac{(k-1)}{k(k+1)} \\ &= 2 \sum_{k=1}^n \left(\frac{2}{k+1} - \frac{1}{k} \right) \\ &= 2 \sum_{k=1}^n \frac{1}{k} - \frac{4n}{n+1}. \end{aligned}$$

Because $\sum_{k=1}^n \frac{1}{k}$ approximates $\int_1^n \frac{1}{x} dx = \ln n$, it follows that

$$\lim_{n \rightarrow \infty} \frac{a_n}{2n \ln n} = 1.$$

Quicksort is, indeed, a very efficient algorithm on average.

2.3. Quickselect

Quickselect [84] is a variation of quicksort designed to find the k th smallest element in an unordered list. For the sake of concreteness, consider the problem of finding an order statistic $x_{(k)}$ from an unsorted array $\{x_1, \dots, x_n\}$ of n distinct numbers. This can be accomplished in $O(n)$ operations based on the quicksort strategy. After the initial partitioning step, one can tell which of the two subarrays contains $x_{(k)}$ just by looking at their sizes. If the left array has $k-1$ entries, then the splitting value is $x_{(k)}$. If the left array has k or more entries, then it contains $x_{(k)}$. Otherwise, the right array contains $x_{(k)}$. Here is Julia code implementing quickselect.

```
function quickselect(x::Vector, k::Int, left = 1, right = length(x))
    i = rand(left:right) # select a random splitting value
    split = x[i]
    (x[left], x[i]) = (split, x[left])
    i = left
    for j = (left + 1):right # position the splitting value
        if x[j] <= split
            i = i + 1
            (x[i], x[j]) = (x[j], x[i])
        end
    end
    (x[left], x[i]) = (x[i], split)
    j = i - left + 1 # find the order statistic y
```

```

    if k == j
      y = x[i]
    elseif k < j
      y = quickselect(x, k, left, i - 1)
    else
      y = quickselect(x, k - j, i + 1, right)
    end
  return y
end

```

```

k = 8;
x = [5, 4, 3, 1, 2, 8, 7, 6];
xk = quickselect(x, k)
k = 5;
x = ['a', 'c', 'd', 'b', 'f', 'e', 'h', 'g'];
xk = quickselect(x, k)

```

Again an average-case analysis is enlightening. Let b_n denote the expected number of operations to find $x_{(k)}$. We now prove that $b_n \leq cn$ with $c = 4$. In view of the fact that it takes $n - 1$ comparisons to create the left and right subarrays, it is obvious that

$$b_n = n - 1 + \frac{1}{n} \sum_{j=1}^{k-1} b_{n-j} + \frac{1}{n} \sum_{j=k}^n b_{j-1}.$$

We now argue by induction that $b_n \leq cn$. This is certainly true for $n = 1$. Suppose it is true for all $k \leq n - 1$. Since $\sum_{i=1}^m i = \binom{m+1}{2}$, it follows that

$$\begin{aligned} b_n &\leq n - 1 + \frac{c}{n} \sum_{j=1}^{k-1} (n - j) + \frac{c}{n} \sum_{j=k}^n (j - 1) \\ &= n - 1 + \frac{c}{n} \left[n(k - 1) - \binom{k}{2} \right] + \frac{c}{n} \left[\binom{n}{2} - \binom{k - 1}{2} \right] \\ &= n - 1 + \frac{c}{2n} (n^2 + 2nk - 2k^2 - 3n + 4k - 2). \end{aligned}$$

Elementary calculus indicates the last quantity is maximized as a function of k by taking $k = \frac{n}{2} + 1$. Substituting this value for k in the bound for b_n yields the new bound

$$b_n \leq n - 1 + \frac{c}{2n} \left(\frac{3n^2}{2} - n \right),$$

which is less than cn whenever $c \geq 4$.

2.4. Heapsort

Heapsort was invented by Williams [173] and Floyd [57] in 1964. Their construction represents the birth of the heap, a useful data structure in its own right. In heapsort the sequence $\mathbf{x} = (x_1, \dots, x_n)$ to be sorted is arranged in a binary tree. The left side of Figure 2.1 displays the binary tree generated by the sequence $\mathbf{x} = (1, 6, 30, 20, 7, 9, 4, 12, 8)$. Note how the data is symmetrically placed along each row from top to bottom and left to right. The j th row contains 2^j items unless j is the last row. In total there are $\lfloor \log_2 n \rfloor + 1$

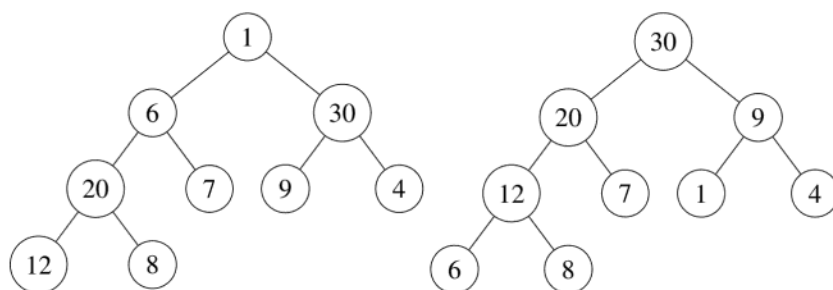


FIGURE 2.1. Transformation of a Binary Sorting Tree into a Heap

rows starting with row 0 at the top. In heapsort the size of the tree gradually diminishes as more items are correctly sorted. The largest item is peeled off first and placed at the end of the sequence. The second largest item is peeled off next and placed adjacent to the largest item, and so forth.

The first phase of heapsort rearranges the items into a heap in which the value of every parent node exceeds the values of its daughter nodes. The left tree in Figure 2.1 shows two violations $1 < \max\{6, 30\}$ and $6 < \max\{20, 7\}$ of this precedence rule. The right tree shows the heap created by systematic rearrangement. A parent node j has daughter nodes $2j$ and $2j + 1$, provided $2j \leq n$ and $2j + 1 \leq n$, respectively. Conversely, the daughters have parent $j = \lfloor \frac{2j}{2} \rfloor = \lfloor \frac{2j+1}{2} \rfloor$. Nodes at the bottom of the tree have no daughters. In phase 2 of the heapsort algorithm, the heap is sorted into increasing order. The two phases of heapsort are reflected in the first function of the following Julia code.

```

function heapsort(x::Vector)
    n = length(x)
    for parent = div(n, 2):-1:1 # form the heap
        siftdown(x, parent, n)
    end
    for bottom = n:-1:2 # sort the heap
        (x[1], x[bottom]) = (x[bottom], x[1])
        siftdown(x, 1, bottom - 1)
    end
end

function siftdown(x::Vector, parent::Int, bottom::Int)
    parent_value = x[parent]
    child = 2 * parent
    while child <= bottom
        if child < bottom && x[child] < x[child + 1]
            child = child + 1
        end
        if x[child] <= parent_value
            break
        else
            x[div(child, 2)] = x[child]
            child = 2 * child
        end
    end
end
  
```

```

    end
    x[div(child, 2)] = parent_value
end

x = [5, 4, 3, 1, 2, 8, 7, 6, -1];
heapsort(x)
println(x)
x = ['a', 'c', 'd', 'b', 'f', 'e', 'h', 'g', 'y'];
heapsort(x)
println(x)

```

The siftdown function of the heapsort code moves a parental value downward in the binary tree until it satisfies the precedence rule. The function first identifies the child of the given parent with the larger value. If this value falls below the parent's value, then the precedence rule is satisfied. Otherwise, the parent's value and the child's value are exchanged, and the child assumes the role of the parent. Again the precedence rule must be checked. The process of exchange and checking continues until the rule is satisfied or the bottom of the tree is reached. When the bottom is reached, the rule is automatically satisfied. In the initial tree of Figure 2.1, the values 1 and 30 are first exchanged. Since $1 < \max\{9, 4\}$, the values 1 and 9 must be exchanged. The algorithm then exchanges 6 and 20. Since $6 < \max\{12, 8\}$, 6 and 12 are exchanged. At this point the tree, corresponding to the sequence (30, 20, 9, 12, 7, 1, 4, 6, 8), is a heap, and the value 30 at the top of the tree is swapped with the value 8 at the bottom-right of the tree.

After 30 is peeled off, the tree is reconstituted with one less node. The new tree corresponds to the reduced sequence (8, 20, 9, 12, 7, 1, 4, 6). Once the precedence violation $8 < 20$ is resolved by a call to siftdown, the value 20 at the top of the new tree can be peeled off. And so it goes until the entire sequence is sorted. The astute reader will notice that resolving the precedence violation caused by a swap affects just a single branch of the tree and restores the heap.

As was noted earlier, heapsort possesses a worst case computational complexity of $O(n \ln n)$. Because the mean of a random variable cannot exceed its maximum value, heapsort also possesses an average computational complexity of $O(n \ln n)$. To verify the worst case behavior, recall that a binary search tree has $\lfloor \log_2 n \rfloor + 1$ rows. Hence, the computational complexity of a single call to siftdown is $O(\ln n)$. Since there are $\lfloor \frac{n}{2} \rfloor$ calls in building a heap, the computational complexity of phase 1 of heapsort is $O(n \ln n)$. The computational complexity of phase 2 is also $O(n \ln n)$ because every time an item is peeled off, siftdown must be called once to restore the heap.

2.5. Bisection

Once an array is ordered, it is easy to search for specific entry values. Fast search depends on the principle of bisection, one of the earliest examples of the divide and conquer strategy in applied mathematics and a dominant theme in search algorithms. In numerical analysis, bisection is used to find the root of a scalar equation $f(x) = 0$. Consider an interval $[a, b]$ where $f(a) < 0 < f(b)$ or $f(a) > 0 > f(b)$. If $f(x)$ is continuous, then the intermediate value theorem guarantees the existence of a root on $[a, b]$. Now let $m = (a + b)/2$ be the midpoint of $[a, b]$. If $f(m) = 0$, then we are done. Otherwise, either $f(a)$ and $f(m)$ are of opposite sign, or $f(b)$ and $f(m)$ are of opposite sign. In the former case, the interval $[a, m]$ brackets a root; in the latter case, the interval $[m, b]$ does. In either case, we replace $[a, b]$ by the corresponding bracketing interval and continue. If we bisect

$[a, b]$ a total of n times, then the final bracketing interval has length $2^{-n}(b - a)$. For n large enough, we can stop and approximate the bracketed root by the midpoint of the final bracketing interval. The following Julia function implements this strategy.

```
function bisect(f::Function, a::T, b::T, tol::T) where T <: Real
    (fa, fb) = (f(a), f(b))
    @assert(a < b && fa * fb <= zero(T)) # check for input error
    for iteration = 1:100
        m = (a + b) / 2
        fm = f(m)
        if abs(fm) < tol
            return (m, iteration)
        end
        if fa * fm < zero(T)
            (b, fb) = (m, fm)
        else
            (a, fa) = (m, fm)
        end
    end
    return ((a + b) / 2, 100)
end
```

```
f(x) = x^3 - 5x + 1.0
(x, iteration) = bisect(f, 0.0, 2.0, 1e-14)
```

The process of bisection extends immediately to searching a discrete ordered list. The only difference is that we have to allow for the possibility that the chosen value does not appear in the list. The following Julia code returns the position of the value in the list. A returned 0 indicates that the value is missing from the list.

```
function binary_search(x::Vector, value)
    a = 1
    b = length(x)
    while a <= b
        m = div(a + b, 2)
        if x[m] > value
            b = m - 1
        elseif x[m] < value
            a = m + 1
        else
            return m
        end
    end
    return 0
end
```

```
x = ['a', 'b', 'd', 'f', 'g'];
println(binary_search(x, 'f'))
x = [1, 2, 4, 7, 9];
println(binary_search(x, 3))
```

2.6. Priority Queues

In computer science, a priority queue is an abstract data type consisting of keys and priorities. Priority queues are typically based on heaps and binary searches. Each key has an associated priority that lives on the heap. Items can be efficiently entered into the queue and extracted according to their priorities. In this section we briefly describe Julia's implementation of priority queues without offering code. Later chapters on graph theory (Dijkstra's algorithm and Prim's algorithm) and linear algebra (Jacobi's algorithm) feature some interesting applications. Julia's priority queues offer three functions: enqueue for inserting a new item, dequeue for deleting the lowest priority item, and peek for exposing the lowest priority item. For example, the commands

```
using DataStructures
```

```
pq = PriorityQueue() # empty queue
pq['a'] = 10 # enqueue or push
pq['b'] = 5
pq['c'] = 15
peek(pq)
dequeue!(pq) # dequeue or pop
```

set up a priority queue with the keys 'a', 'b', and 'c' and the priorities 10, 5, and 15, respectively. The peek command returns ('b', 5). The dequeue! command deletes the pair ('b', 5) from the queue.

2.7. Problems

- (1) What is the probability that a random permutation of n distinct numbers contains at least one preexisting splitter? What are the mean and variance of the number of preexisting splitters?
- (2) Show that the worst case of quicksort takes on the order of n^2 operations.
- (3) When an item is peeled off the top of the heap in heapsort, prove formally that at most a single branch of the new tree must be subjected to sift-down to restore the tree to a heap.
- (4) Given a sorted array of numbers of length n and a number c , write a Julia program to find the pair of numbers in the array whose sum is closest to c . An efficient solution can find the pair in $O(n)$ time.
- (5) Suppose \mathbf{x} and \mathbf{y} are two sorted arrays of numbers of length m and n , respectively. Design and implement a Julia program to merge \mathbf{x} and \mathbf{y} into a single sorted array of length $m + n$. Your algorithm should have computational complexity $O(m + n)$.
- (6) Suppose you are given two ordered arrays \mathbf{x} and \mathbf{y} of integers. If these represent integer sets, write Julia functions to find their union, intersection, and set difference. Do not use existing Julia functions for set operations.
- (7) Two entries x_i and x_j of a numerical sequence $\mathbf{x} = (x_1, \dots, x_n)$ represent an inversion if $x_i > x_j$ and $i < j$. Write an efficient Julia function to count the number of inversions in \mathbf{x} .
- (8) Write a bisection algorithm to find the quantiles of a gamma distributed random variable.
- (9) Write an efficient Julia function to extract the closest entry of a ordered numerical sequence \mathbf{x} to a given number c . (Hint: Use bisection.)

- (10) Counting sort assumes that all items in a sequence belongs to a small list of k existing items. Design and implement a Julia function for counting sort with computational complexity $O(n)$.
- (11) Bucket sort assumes that the entries of a sequence x are drawn independently and uniformly from the interval $[0, 1]$. Design and implement a Julia function for bucket sort with computational complexity $O(n)$ on average.

Graph Algorithms

3.1. Introduction

Graph theory was initiated by Leonhard Euler in his 1736 study of the Seven Bridges of Königsberg problem [12]. In the intervening centuries, graph theory has blossomed into one of the most fertile branches of discrete mathematics. The applications in computer science, linguistics, physics, chemistry, the social sciences, biology, and various branches of applied mathematics are simply too numerous to adequately summarize here. The modern synonym network suggests some of the potential of graph theory.

Graphs are mathematical structures consisting of nodes and edges. An edge connects two nodes. Sometimes nodes are referred to as vertices or points and edges as arcs or lines. A graph may be undirected or directed; in the latter case each edge has an orientation and points from its tail to its head. We will reserve the term graph for undirected graphs and call directed graphs digraphs. In a weighted graph or digraph, each edge is assigned a nonnegative weight. The adjacency matrix of the graph or digraph encodes these weights. The absence of an edge is indicated by a zero entry of the matrix. This matrix is symmetric for a graph and consists entirely of 0/1 entries in the absence of weights. The neighbors of a node are those immediately adjacent to it. The number of neighbors of node i in a graph or digraph is called the degree of i .

The digraph of a discrete-time Markov chain has edge weights giving the probabilities of moving from one state to the next [56]. The digraph of a chain is unusual in the sense that the tail and head of an edge can coincide. This occurs when the chain can remain in place for a random number of epochs (generations). The weighted adjacency matrix $\mathbf{P} = (p_{ij})$ of a Markov chain is referred to as a probability transition matrix. Its row sums $\sum_j p_{ij}$ equal 1. Section 8.3 explores the power method for finding the equilibrium distribution of a Markov chain.

This chapter focuses on four fundamental algorithms of graph theory, roughly in order of subtlety [35]. The first shows how to pass from the adjacency matrix description of a graph or digraph to its neighborhood description. The second collects the nodes of a graph into connected components. The third, Dijkstra's algorithm, finds the shortest path from a source node to every other node of a weighted graph or digraph. The fourth, Prim's algorithm, finds a minimum spanning tree of a weighted graph. The latter two algorithms operate in a greedy fashion. These successes are exceptions to the rule of thumb that most greedy algorithms are undone by their shortsightedness.

3.2. From Adjacency to Neighborhoods

The graph depicted in Figure 3.1

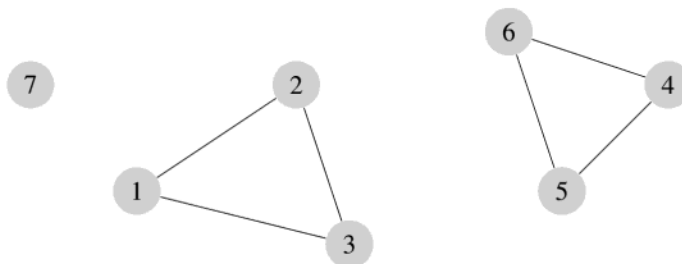


FIGURE 3.1. A Graph with 7 Nodes and 3 Components

has the symmetric adjacency matrix

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Here and in the sequel, we adopt the convention that the nodes of a graph or digraph with n nodes are numbered 1 through n . The following Julia code converts \mathbf{A} into a system of neighborhoods and weights stored as two vectors of vectors.

```
function adjacency_to_neighborhood(A::AbstractMatrix)
    (nodes, T) = (size(A, 1), eltype(A))
    neighbor = [Vector{Int}() for i = 1:nodes]
    weight = [Vector{T}() for i = 1:nodes]
    for i = 1:nodes
        for j = 1:nodes
            if A[i, j] != zero(T)
                push!(neighbor[i], j)
                push!(weight[i], A[i, j])
            end
        end
    end
    return (neighbor, weight)
end
```

```
A = [[ 0 1 1 0 0 0 0]; [ 1 0 1 0 0 0 0]; [ 1 1 0 0 0 0 0];
      [ 0 0 0 0 1 1 0]; [ 0 0 0 1 0 1 0]; [ 0 0 0 1 1 0 0];
      [ 0 0 0 0 0 0 0]];
(neighbor, weight) = adjacency_to_neighborhood(A);
```

The code is more or less self-explanatory. On output the j th neighbor of node i is accessed as `neighbor[i][j]`. If k is this neighbor, then the corresponding weight of the edge (i, k) is stored in `weight[i][j]`. The command `length(neighbor[i])` delivers the degree of node i . If the matrix \mathbf{A} enters the function with real entries, then the entries of `weight` are real-valued rather than integer-valued.

Exactly the same code is applicable to digraphs. In this setting the command `length(neighbor[i])` delivers the number of nodes reachable from i in one step. This number is called the out-degree of i . In-degrees of each node can be computed from the transpose of the adjacency matrix.

3.3. Connected Components

In graph theory a path is a finite sequence of nodes $a - b - c - \dots$ connected by consecutive edges. Some authors further stipulate that no node should be repeated. We omit this extra requirement. In a graph two nodes are connected if there is a path between them. The relation of pathwise connectedness splits the nodes of a graph into equivalence classes. Recall that an equivalence relation \equiv is characterized by the properties (a) $a \equiv a$ (reflexive), (b) $a \equiv b \implies b \equiv a$ (symmetry), and (c) $a \equiv b$ and $b \equiv c \implies a \equiv c$ (transitive). In a graph a maximal equivalence class is called a component. For example, the graph of Figure 3.1 has the 3 components $\{1, 2, 3\}$, $\{4, 5, 6\}$, and $\{7\}$. A digraph splits into strongly connected components. Within a component, each node is reachable from every other node of the component. In a digraph a path $a \rightarrow b \rightarrow c \rightarrow \dots$ shows a consistent orientation of consecutive edges. Furthermore, the existence of a path $a \rightarrow \dots \rightarrow b$ does not necessarily imply the existence of a path $b \rightarrow \dots \rightarrow a$ in the reverse direction. Reachability is by definition bidirectional.

Fortunately, there is a simple depth-first algorithm for finding the components of a graph [83]. A depth-first search visits a node's children before it visits the node's siblings. A breadth-first search operates in the opposite fashion. The following Julia code incorporates two functions, the second of which recursively performs the depth-first search. The first function delivers the number of components and an assigned component for each node.

```
function connect(neighbor::Array{Array{Int, 1}, 1})
    nodes = length(neighbor)
    component = zeros{Int, nodes}
    components = 0
    for i = 1:nodes
        if component[i] > 0 continue end
        components = components + 1
        component[i] = components
        visit!(neighbor, component, i)
    end
    return (component, components)
end

function visit!(neighbor::Array{Array{Int, 1}, 1},
               component::Vector{Int}, i::Int)
    #
    for j in neighbor[i]
        if component[j] > 0 continue end
        component[j] = component[i]
        visit!(neighbor, component, j)
    end
end

A = [[ 0 1 1 0 0 0 0]; [ 1 0 1 0 0 0 0]; [ 1 1 0 0 0 0 0]];
```

```
[ 0 0 0 0 1 1 0]; [ 0 0 0 1 0 1 0]; [ 0 0 0 1 1 0 0];
[ 0 0 0 0 0 0 0];
(neighbor, weight) = adjacency_to_neighborhood(A);
(component, components) = connect(neighbor)
```

There exist similar but more complicated algorithms for finding the strongly connected components of a digraph [155, 163].

3.4. Dijkstra's Algorithm

Dijkstra's algorithm is designed to find the shortest paths from a source node s in a weighted graph to all other nodes i in the graph [48]. As noted earlier, the algorithm is greedy and also applies to weighted digraphs. Remarkably, it is about as simple to find all shortest paths as it is to find a single shortest path. The following two results are of independent interest and used implicitly in proving the correctness of Dijkstra's algorithm.

PROPOSITION 3.4.1. *A subpath of any shortest path beginning at the source is itself a shortest path. If $d(i, j)$ denotes the shortest weighted path length between two nodes i and j of a graph or digraph, then $d(i, j)$ obeys the triangle inequality*

$$d(i, k) \leq d(i, j) + d(j, k).$$

Proof: This task is relegated to problem (2). □

Dijkstra's algorithm is an example of dynamic programming. It solves a sequence of successively larger subproblems that converges after a finite number of steps to the full problem. Dijkstra's algorithm adds one node at a time to a growing subgraph G_n . In the process it updates a vector \mathbf{d}_n of provisional shortest distances from the source s . By a clever construction, the components d_{ni} of \mathbf{d}_n for $i \in G_n$ turn out to be true shortest distances. For a node i neighboring the subgraph G_n , d_{ni} is the shortest distance from s to i through G_n . For all other nodes $d_{ni} = \infty$. The provisional distances are stored in a priority queue [58]. As just described, d_{ni} decreases as n increases and is fixed at its true value d_i when i is visited. At that point d_i is removed from the priority queue. As the algorithm proceeds, a predecessor node is recorded for each node popped off the queue. The predecessors permit reconstruction of the shortest paths. If a node is beyond the reach of the source, its distance is returned as ∞ . With this outline in mind, here is Julia code implementing Dijkstra's algorithm.

using DataStructures

```
function dijkstra(neighbor::Array{Array{Int, 1}, 1},
  weight::Array{Array{T, 1}, 1}, source::Int) where T <: Number
#
  nodes = length(neighbor)
  node = collect(1:nodes) # the nodes are numbered 1, 2, ...
  predecessor = zeros{Int, nodes}
  visited = falses{nodes}
  distance = zeros{nodes}
  fill!(distance, Inf)
  distance[source] = 0.0
  pq = PriorityQueue{zip{node, distance}} # priority queue
  while !isempty(pq)
    (i, d) = peek(pq) # retrieve the minimum remaining distance
```

```

distance[i] = d
visited[i] = true
dequeue!(pq) # pop the current node
for k = 1:length(neighbor[i])
    j = neighbor[i][k]
    if !visited[j]
        dij = d + weight[i][k]
        if pq[j] > dij
            predecessor[j] = i
            pq[j] = dij # adjust the provisional distance to j
        end
    end
end
end
return (distance, predecessor)
end

```

```

A = [[ 0 7 9 0 0 14]; [ 7 0 10 15 0 0]; [ 9 10 0 11 0 2];
[ 0 15 11 0 6 0]; [ 0 0 0 6 0 9]; [ 14 0 2 0 9 0]];
(neighbor, weight) = adjacency_to_neighborhood(A);
(distance, predecessor) = dijkstra(neighbor, weight, 1)

```

The displayed Dijkstra code is applied to the graph with adjacency matrix

$$A = \begin{pmatrix} 0 & 7 & 9 & 0 & 0 & 14 \\ 7 & 0 & 10 & 15 & 0 & 0 \\ 9 & 10 & 0 & 11 & 0 & 2 \\ 0 & 15 & 11 & 0 & 6 & 0 \\ 0 & 0 & 0 & 6 & 0 & 9 \\ 14 & 0 & 2 & 0 & 9 & 0 \end{pmatrix}.$$

The shortest paths identified from 1 as source are $1 \rightarrow 2$, $1 \rightarrow 3$, $1 \rightarrow 3 \rightarrow 4$, $1 \rightarrow 3 \rightarrow 6 \rightarrow 5$, and $1 \rightarrow 3 \rightarrow 6$. The next proposition proves the correctness of Dijkstra's algorithm.

PROPOSITION 3.4.2. *For a graph or digraph with m nodes, Dijkstra's algorithm terminates after m steps with the minimal distances.*

Proof: The correctness of Dijkstra's algorithm can be verified by induction on the number n of visited nodes. For brevity, denote the edge weights by w_{ij} . Let G_n denote the subgraph defined by the visited nodes at stage n . The induction hypothesis states that for every $i \in G_n$, d_{ni} equals the length of a shortest path from the source s to i . This shortest distance may be ∞ . Because d_{1s} is initialized as 0, the induction starts correctly. Assume the hypothesis is true for $n - 1$ visited nodes. The algorithm now chooses an unvisited node i whose distance $d_{n-1,i}$ is least. At this juncture, i is declared visited, and d_{ni} is set equal to $d_{n-1,i}$. Furthermore, the distance $d_{n-1,j}$ to each unvisited neighbor j of i is checked to see whether it should be revised by taking a path passing through i . This is required when $d_{n-1,i} + w_{ij} < d_{n-1,j}$. To complete the inductive argument, we show that the distance $d_{ni} = d_{n-1,i}$ is minimal. Our argument compares a shortest path to i with the shortest path through G_{n-1} to i . The second path has length $d_{n-1,i}$. The first path initially passes through G_{n-1} and exits it to some node j outside G_{n-1} . It must then traverse an additional subpath from j to i . The total distance of the two paths are $d_{n-1,i}$

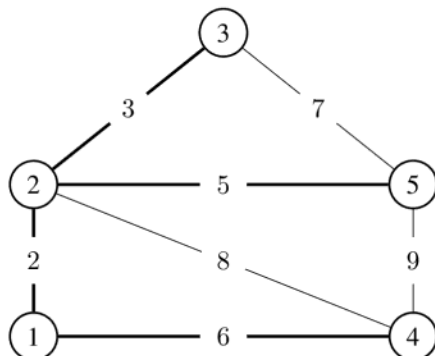


FIGURE 3.2. Minimum Spanning Tree of a Connected Graph

and $d_{n-1,j} + c_{ji}$, where c_{ji} is the length of the subpath from j to i . The inequalities $c_{ji} \geq 0$ and $d_{n-1,i} \leq d_{n-1,j}$ now prove our assertion and complete the induction. \square

3.5. Prim's Algorithm

Prim [144], Jarník [89], and Dijkstra [48] share the credit for this basic graph algorithm. It is invoked by a network router to minimize the routes to other components in a network. A tree is a connected graph with no superfluous edges. In other words, deleting any edge disconnects the tree. Because of the assigned edge weights, trees may vary widely in their edge weight sums. Fortunately, a greedy algorithm discovers the minimum spanning tree (MST). Figure 3.2 depicts the MST of its weighted graph by thick edges.

The overall algorithm bears a strong resemblance to Dijkstra's shortest path algorithm. Each node is tracked as visited or not yet visited. At stage n of the algorithm, an edge of minimum weight is added to the current subtree T_{n-1} to produce an enlarged subtree T_n with one more node. One node of the edge is attached to T_{n-1} , and one node falls outside T_{n-1} . Deleting this edge or any previously added edge disconnects T_n .

A priority queue again plays a critical role. However, the keys of the priority queue are now edges rather than nodes, and priorities are weights rather than distances. Edges are popped off the queue as they are added to the minimum spanning tree or connect two already visited nodes. The tree grows from its tip, which is updated as the algorithm progresses. The edges neighboring an unvisited tip are pushed onto the queue. The tree itself is recorded as a sequence of edges. The code for Prim's algorithm follows.

```
using DataStructures
```

```
function prim(neighbor::Array{Array{Int, 1}, 1},
  weight::Array{Array{T, 1}, 1}) where T <: Number
#
  nodes = length(neighbor)
  visited = falses(nodes)
  (mst_nodes, i) = (1, 1) # initialize MST with node 1 as tip
  key = Array{Tuple{Int, Int}, 1}(undef, 0) # define keys
  priority = Array{Float64, 1}(undef, 0) # define priorities
  pq = PriorityQueue{zip(key, priority)} # initialize queue
  mst = Array{Tuple{Int, Int}, 1}(undef, 0) # minimum spanning tree
```

```

while mst_nodes < nodes
  if !visited[i] # checked if the node has been visited
    visited[i] = true # mark the node as visited
    for k = 1:length(neighbor[i]) # add new edges to the queue
      j = neighbor[i][k]
      pq[(i, j)] = weight[i][k]
    end
  end
  ((k, l), val) = peek(pq) # choose lightest edge
  dequeue!(pq) # pop edge off queue
  if !visited[k] # if not part of tree, push the edge onto tree
    push!(mst, (k, l))
    mst_nodes = mst_nodes + 1
    i = k # k is the new tip
  elseif !visited[l]
    push!(mst, (k, l))
    mst_nodes = mst_nodes + 1
    i = l # l is the new tip
  end
end
return mst
end

```

```

A = [0 2 0 6 0; 2 0 3 8 5; 0 3 0 0 7; 6 8 0 0 9; 0 5 7 9 0];
(neighbor, weight) = adjacency_to_neighborhood(A);
mst = prim(neighbor, weight)

```

The correctness of Prim's algorithm stems from a simple property of spanning trees and cycles. Recall that a cycle is a path that starts and ends with the same node.

PROPOSITION 3.5.1. *Suppose we take a spanning tree T and add any edge not in T to it. This action creates a cycle. If we remove any edge from the cycle, then we are left with a possibly new spanning tree.*

Proof: This proof is relegated to problem (5). □

PROPOSITION 3.5.2. *For a weighted connected graph, Prim's algorithm terminates after a finite number of steps with a minimal spanning tree.*

Proof: We first prove that Prim's algorithm constructs a spanning tree. It starts with a tree T_1 with a single node. Suppose at stage $n-1$ it produces a tree T_{n-1} . At stage n it adds an edge e that has one node in T_{n-1} and one node outside T_{n-1} . Hence, $T_n = T_{n-1} \cup \{e\}$ also is a tree. When it exhausts all nodes, Prim's algorithm therefore yields a spanning tree T . If the tree T is not minimal, let n be the first stage at which T_{n-1} can be embedded in a minimal spanning tree S , but T_n cannot. Hence, the edge e added in going from T_{n-1} to T_n falls outside S . Proposition 3.5.1 implies that $S \cup e$ contains a cycle. Because e has one endpoint in T_{n-1} and one endpoint outside T_{n-1} , there must be another edge f along the cycle with exactly one endpoint in T_{n-1} . The assumption $w_e \leq w_f$ is dictated by Prim's algorithm. Thus, we can delete f from S and add e and still wind up with a minimal spanning tree. This contradicts our assumption that T_n is inconsistent with a minimal spanning tree. □

3.6. Problems

- (1) Prove that if x is a node of a graph having odd degree, then there exists a path from x to another node y of the graph having odd degree.
- (2) Prove Proposition 3.4.1.
- (3) Consider a connected graph with n nodes. Demonstrate that it is a tree if and only if it has $n - 1$ edges.
- (4) Prove that between any two nodes of a tree, there is one and only one connecting path.
- (5) Prove Proposition 3.5.1.
- (6) If all of the edge weights of a connected graph are unique, then prove that the graph has exactly one minimum spanning tree.
- (7) Let A be the adjacency matrix of a graph. Let the n th power have entries a_{nij} . Show that a_{nij} equals the number of paths from i to j with exactly n steps. Verify your result empirically for a simple graph.
- (8) Prove that Dijkstra's algorithm has running time $O(E \log N)$, where E is the number of edges and N is the number of nodes.
- (9) Prove that Prim's algorithm has running time $O(E \log N)$, where E is the number of edges and N is the number of nodes.
- (10) Program either Kosaraju's algorithm [155] or Tarjan's algorithm [163] for finding the strongly connected components of a digraph.
- (11) Program Kruskal's algorithm [104] for finding the minimum spanning tree of a weighted graph.

Primality Testing

4.1. Introduction

Two generations ago number theory was considered one of the purest branches of pure mathematics. It has now been sullied by applications to high-speed arithmetic on computers, numerical analysis in general, cryptography, and especially secure communication. Prime numbers have been the obsession of number theorists since Euclid. These mysterious entities are seemingly scattered at random throughout the natural numbers. Yet primes show some surprising regularities. For instance, the prime number theorem says the number of primes less than or equal to n is asymptotic to $\frac{n}{\ln n}$. There are many unresolved questions about primes. The twin prime conjecture deals with primes n and $n + 2$. Is there an infinite sequence of such primes? The empirical evidence overwhelmingly supports the conjecture, but no one has proved it.

In this chapter we take up the simpler problem of determining whether a number n is prime. The Sieve of Eratosthenes solves the problem when n is small. However, the sieve is far too cumbersome for the large primes of current interest. The Miller–Rabin algorithm [130, 145] studied here is fast, straightforward to implement, and inherently probabilistic. There are deterministic algorithms for the same purpose, but they are more complicated to understand and much slower in practice [1, 149]. Unfortunately, the Miller–Rabin does not deliver the prime factorization of a composite number. This is a much harder problem. Readers who master the material in this chapter will be in a good position to tackle the RSA (Rivest–Shamir–Adleman) algorithm [151] for data encryption. The RSA algorithm works because factoring composite numbers is so challenging.

Appendix A.2 is intended to bring readers up to speed on elementary number theory. Readers will also benefit from the enormous array of books on number theory. We especially recommend the classics [5, 72, 135]. These are bound to stimulate your appetite for this beautiful and arcane branch of mathematics.

4.2. Perfect Powers

We begin our exposition of primality testing by tackling a much simpler problem. A positive integer n is a perfect power if it can be expressed as k^j for integers $j > 1$ and $k > 1$. One can design a fast algorithm for testing this property by relying on two crucial insights. First, $j \leq \log_2 n$ owing to the inequality $2^j \leq k^j \leq n$ for all relevant j and k . Second, the exponent j can be further restricted to the set of prime numbers. Indeed, if $n = k^j$ and $j = pq$ with p prime, then $(k^q)^p = n$. To isolate the pertinent primes, we can call on the Sieve of Eratosthenes discussed earlier. For a given prime j , the only possible k is $\sqrt[j]{n}$. Here is Julia code implementing the perfect power algorithm.

```
function perfectpower(n::Integer)
    m = Int(floor(log(2, n))) # integer part of log base 2 of n
    prime_list = eratosthenes(m)
```

```

for j in prime_list
  k = Int(round(n^(1 / j)))
  if isequal(k^j, n)
    return true
  end
end
return false
end

```

perfectpower(1000)

In the code $\sqrt[j]{n}$ is calculated as $e^{\frac{1}{j} \ln n}$. Example 6.2 sketches a better algorithm for extracting roots based on Newton's method. The Babylonian method for extracting square roots is a special case.

4.3. Modular Arithmetic and Group Theory

Readers unfamiliar with elementary number theory should review Appendix A.2 at this juncture. Here one finds the definition of the algebraic structure \mathbb{Z}_n . This object is just the set of integers $\{0, 1, \dots, n-1\}$ equipped with modular addition and modular multiplication. If a and b belong to \mathbb{Z}_n , then $a + b \bmod n$ and $ab \bmod n$ are defined as the remainders of $a + b$ and ab on division by n . Note that \mathbb{Z}_n inherits the usual commutative and associative laws of arithmetic. As expected, the additive and multiplicative identities of \mathbb{Z}_n are 0 and 1, respectively. Division is not always possible. If $\gcd(a, n) = 1$, then there exists a reciprocal a^{-1} such that $a^{-1}a = 1 \bmod n$. In the special case where n is prime, a^{-1} exists for all $a \neq 0 \bmod n$, and \mathbb{Z}_n is algebraically a field.

The set of integers $u \in \mathbb{Z}_n$ with $\gcd(u, n) = 1$ is denoted \mathbb{U}_n . The integers in \mathbb{U}_n are sometimes referred to as units; \mathbb{U}_n is closed under multiplication and contains 1 and $-1 = n-1 \bmod n$. Indeed, if u_1 and u_2 share no nontrivial divisors with n , then their product u_1u_2 also shares no nontrivial divisors with n . The reciprocal of every unit u is also a unit. This fact is proved in Proposition A.2.7. One can summarize our findings by noting that \mathbb{U}_n constitutes a finite commutative group.

We will need to dip briefly into the theory of finite groups. A group G is a set such as \mathbb{U}_n equipped with multiplication and possessing an identity element 1 satisfying $a1 = 1a = a$ for all $a \in G$. Furthermore, every group element $a \in G$ has a left inverse b satisfying $ba = 1$. Regardless of whether G is commutative, one can show that a left inverse is also a right inverse. The order of a finite group G is just its cardinality $|G|$. A subgroup H of a group G is a nonempty subset of G closed under multiplication and the formation of reciprocals (inverses). A subgroup automatically contains the identity element 1 of the group. We will need the following two properties of a subgroup, the second of which is known as Lagrange's theorem.

PROPOSITION 4.3.1. *If a nonempty subset H of a finite group G contains 1 and is closed under multiplication, then H is a subgroup. The order $|H|$ of a subgroup H divides the order $|G|$ of the group.*

Proof: To prove the first assertion, we must show that the inverse a^{-1} of $a \in H$ belongs to H . Consider the map $H \mapsto H$ defined by $b \mapsto ab$. This map is one-to-one because if $ab = ac$, then $b = c$. Since H is finite, the map is also onto. Hence, there exists $b \in H$ with $ab = 1$. For the second assertion, we introduce the notion of a left coset. Consider the map $b \mapsto ab$ for a not necessarily in H . It maps H into a set aH . Regardless

Index

- ABO algorithm, 117
- acceptance function, 176
- acceptance-rejection method, 163–166
- active constraint, 189
- adjacency matrix, 21–23
- algorithm
 - ABO, 117
 - ANOVA, 111–113
 - Archimedes, 5–8
 - Babylonian, 2–3
 - bisection, 16–17
 - Cholesky decomposition, 41–44
 - conjugate gradient, 47–50
 - convolution, 148
 - cosine minimization, 109
 - Dedekind's, [9](#)
 - Dijkstra's, 24–26
 - Dirichlet distribution, 110–111
 - discrete deviate, 162
 - EM clustering, 123–126
 - Euclid's, 4–5
 - fast Fourier transform, 143–145
 - Fisher scoring, 67
 - gamma deviate, 164
 - gamma-Poisson, 118
 - Gaussian deviate, 163
 - Gaussian elimination, 37–41
 - Gibbs sampling, 172
 - Goldschmidt's, [8](#)
 - Gram–Schmidt orthogonalization, 44–47
 - graph components, 23–24
 - graph neighborhoods, [21](#)
 - Halley's, 68
 - hardcore model, 168
 - heapsort, 14–16
 - Hestenes and Karush, 103
 - Jacobi's, 96–100
 - Karmarkar, 78–81
 - Lloyd's, 121–123
 - LU decomposition, 37–41
 - matrix square root, 69
 - median, 117
 - Miller–Rabin, 32–34
 - modular exponentiation, 31
 - naive Bayes, 126–128
 - Nash's, 100
 - negative binomial, 109–110
 - Newton's, 53–70
 - nonnegative matrix factorization, 133–137
 - PageRank, 88
 - peasant multiplication, [1](#)
 - perfect power, [30](#)
 - Poisson deviate, 170
 - positron tomography, 113–116
 - power series distribution, 118
 - Prim's, 26–27
 - QR decomposition, 44–47, 93
 - quadratic formula, 3–4
 - quickselect, 13–14
 - quicksort, 11–13
 - random number generation, 159–160
 - Rayleigh quotient, 89
 - revised simplex, 74–76
 - Sieve of Eratosthenes, [5](#)
 - singular value decomposition, 100
 - traveling salesman, 173
 - Weibull distribution, 119
 - Zipf deviate, 165
- analytic function, 146–148
- ANOVA, 111–113
- Archimedes' approximation of π , 5–9
- arcsine distribution, 175
- Armijo–Goldstein test, 63
- asymptotic functions, 179
- autocovariance, 153
- average-case analysis, [12](#), [14](#), [16](#)
- Babylonian method, 2–3, [8](#), 55
- backward substitution, 40, 43
- Banachiewicz, Tadeusz, 37
- banded matrix, 51
- Bayes' rule, 123, 126
- Bezout's identity, [5](#), [9](#), 35
- big oh, 179
- binary tree, [15](#)
- binomial theorem, 69
- bisection, 16–17
- bit reversal, 153

- bit shifting, [1](#)
- BLAS, [37](#)
- branching process, [146](#)
- bucket sort, [19](#)
- cache memory, [42](#)
- casting out nines, [34](#)
- Cauchy–Schwarz inequality, [187](#), [188](#), [193](#), [195](#)
- Cayley transform, [102](#)
- central limit theorem, [164](#)
- change point problem, [171](#)
- Chinese remainder theorem, [33](#), [34](#), [182](#)
- Cholesky decomposition, [41–44](#), [51](#)
 - banded matrix, [51](#)
- circulant matrix, [151](#)
- classification
 - k -nearest neighbors, [128–129](#)
 - matrix completion, [130–133](#)
 - naive Bayes, [126–128](#)
- cluster analysis
 - EM, [123–126](#)
 - k -means, [121–123](#)
- clustering
 - EM algorithm, [137](#)
- coin-tossing, waiting time, [156](#)
- complexity levels, [179](#)
- conjugate gradient algorithm, [47–50](#)
- conjugate vectors, [48](#), [52](#)
- contraction mapping theorem, [87](#), [201](#)
- convex function
 - closure properties, [187](#)
 - epigraph test, [106](#)
 - Jensen’s test, [106](#)
 - minimum, [188](#)
 - second derivative test, [106](#), [186](#)
 - strong convexity, [188–189](#)
 - supporting hyperplane, [106](#), [186](#)
- convex programming, [190](#)
- convex set, [82](#), [185](#)
- convolution
 - data smoothing, [149](#)
 - finite differencing, [149](#)
 - integer multiplication, [150](#)
 - periodic sequences, [142](#), [148–151](#)
- coordinate descent, [47](#)
- coprime numbers, [182](#)
- cosine minimization, [108](#)
- cubic equation, [9](#)
- Dantzig, George, [71](#)
- data compression, [133](#)
- data mining, [121–140](#)
- data smoothing, [149](#)
- Davidon’s rank-one update, [67](#)
- Dedekind’s algorithm, [9](#)
- depth-first search, [23](#)
- Descartes, René, [3](#)
- descent direction, [63](#)
- descent property, [105](#)
- determinant, [41](#), [42](#), [193](#)
- diet problem, [71](#)
- differential, [xi](#)
- differentiation
 - analytic functions, [146–148](#)
- digraph, [21](#)
 - path, [23](#)
- Dijkstra’s algorithm, [24–26](#), [28](#)
- Dinkelbach maneuver, [79](#)
- Dirichlet distribution, [110–111](#)
- discrete Fourier transform, [141–158](#)
 - calculation rules, [142](#)
 - convolution, [148–151](#)
 - definition, [141](#)
 - examples, [154–155](#)
 - fast Fourier transform, [143–145](#)
 - Fourier series approximation, [145–148](#)
 - inversion, [141](#)
 - periodic sequence, [142–143](#)
 - renewal equations, [150](#)
 - time series, [153–154](#)
- discriminant analysis, [130–133](#), [138](#)
- dominated convergence theorem, [154](#)
- Eckart–Young theorem, [139](#), [199](#)
- eigenvalues and eigenvectors, [85](#)
 - dominant, [103](#)
 - examples, [102](#)
 - symmetric matrix, [194](#)
- eigenvector, [194](#)
- EM algorithm, [106](#)
- EM clustering, [123–126](#), [137](#)
- equality constraint, [189](#)
- equivalence relation, [23](#)
- Erdős, Paul, [ix](#)
- ergodic theorem, [168](#), [176](#)
- Euclid’s algorithm, [4–5](#)
- Euclidean division, [180](#)
- Euclidean norm, [xi](#)
- Euclidean projection, [44–45](#), [80](#), [135](#), [140](#)
- Euler’s theorem, [32](#)
- Euler’s totient, [32](#), [35](#)
- extreme point, [82](#)
- fast Fourier transform, [143–145](#)
 - code, [144–145](#)
 - computational complexity, [144](#)
- feasible point, [189](#)
- Fermat’s little theorem, [31](#), [32](#), [35](#), [160](#), [183](#)
- finite differencing, [149](#)
- finite field, [182](#)
- Fisher’s iris data, [132](#)
- Floyd, Robert, [14](#)
- forward substitution, [40](#), [43](#)
- Fourier coefficients
 - approximation, [145–148](#)
- Frobenius inner product, [140](#)
- Frobenius norm, [xi](#), [69](#), [129](#), [134](#), [139](#), [140](#), [195](#), [196](#)

- function
 - convex, 186–189
 - differentials, xi
 - digamma, 110
 - objective, 189
 - quadratic, 47
 - Rosenbrock's, 117
- fundamental theorem
 - arithmetic, 181
 - calculus, 62
 - linear algebra, 80, 193
 - linear programming, 74–75
- gamma distribution, 171
- gamma-Poisson distribution, 118
- Gauss–Newton algorithm, 65–66
- Gaussian distribution, 44, 123–125, 175
- Gaussian elimination, 37–41
- generalized linear model, 67
- generating function
 - branching process, 146
 - coin-toss wait time, 156
 - multiplication, 149
- Gibbs sampling, 171–173
- Goldschmidt's algorithm, [8](#)
- gradient, xi
- Gram–Schmidt orthogonalization, 44–47, 51
- graph, [21](#)
 - connected components, 23–24
 - cycle, [27](#)
 - path, [23](#)
 - strongly connected components, [28](#)
 - tree, [26](#), [28](#)
- graph bisection, 177
- greatest common divisor, [4](#), [9](#), [30](#), 181
- greedy algorithm, [21](#), 98
- group
 - additive, 180
 - Lagrange's theorem, [30](#)
 - subgroup, [30](#)
- Gumbel distribution, 175
- Hadamard transform, 157
- Halley's method, 68
- Hankel matrix, 152
- heap, [15](#)
- heapsort, 14–16
- Hessian, xi
- Hestenes and Karush algorithm, 103
- Hestenes, Magnus, 47
- Hoare, Anthony, [11](#), [13](#)
- Horner's method, [10](#)
- Householder transformation, 90–93
- hyperbolic trigonometric functions, 156
- inactive constraint, 189
- indicator of a set, xi
- induced matrix norm, 195
- inequality
 - Cauchy–Schwarz, 187, 188, 193, 195
 - Jensen's, 106, 114, 186, 189
 - supporting hyperplane, 186
 - triangle, [24](#), 195
- inequality constraint, 189
- instrumental density, 163
- intermediate value theorem, [16](#)
- inverse method, 160–161, 175
- inverse Wishart distribution, 137
- inversion, combinatorial, [18](#)
- Jacobi's method, 96–100
- Jensen's inequality, 106, 114, 186, 189
- k*-means clustering, 121–123, 137
- k*-nearest neighbors, 128–129, 137
- Kantorovich, Leonid, 71
- Karmarkar's algorithm, 78–81
- Karush–Kuhn–Tucker theorem, 189
- Koopmans, Tjalling, 71
- Lagrange multipliers, 189–193
- Lagrange's theorem, [30](#), 34
- Lagrangian
 - linear programming, 74
 - multinomial probabilities, 191
 - quadratic program, 192
- Lambert's equation, 67
- LAPACK, 37
- large integer multiplication, 150
- law of large numbers, 168
- least squares
 - convexity, 188
 - linear, 43
 - Moore–Penrose inverse, 199
 - Newton's method, 61
 - nonlinear, 65–66
 - weighted, 69
- Levenberg–Marquardt method, 63
- linear equations, 37
- linear fractional programming, 73
- linear logistic regression, 63–65
- linear programming, 71–81
 - applications, 71–74, 81
 - Bland's rules, 75
 - canonical form, 71
 - complementary slackness, 75
 - dual program, 82
 - homogeneous form, 78
 - initialization, 76
 - optimal basic feasible point, 74
 - revised simplex method, 74–76
- linear regression, 43–45
- little oh, 179
- Lloyd's algorithm, 121–123
- logistic distribution
 - sampling, 174
- lower triangular matrix, 38
- LU decomposition, 37–41

- majorization
 - definition, 105
 - Jensen's, 107
 - log splitting, 107
 - negative logarithm, 117
 - product, 116
 - quadratic bound, 107
- Markov chain, [21](#), 86–89
 - equilibrium distribution, 87
 - reversible, 167
- matrix
 - banded, 51
 - Cholesky decomposition, 41, 124
 - circulant, 151
 - completion, 129–130
 - computation of powers, 31
 - determinant, 41, 42
 - diagonalizable, 87
 - factorization, 152
 - fast multiplication, 50
 - Hankel, 152
 - inversion, 41, 51, 57
 - lower triangular, 38, 124
 - norm, 195
 - orthogonal, 51, 152, 196
 - permutation, 39, 51
 - positive definite, 103–104
 - QR decomposition, 44–47, 52
 - randomized multiplication, 166–167
 - reflection, 196
 - rotation, 196
 - skew-symmetric, 102
 - sparse, 47, 76, 131
 - spectral decomposition, 194
 - square root, 69
 - Toeplitz, 152
 - tridiagonal, 51
 - upper triangular, 38, 51
- matrix inversion
 - Moore–Penrose inverse, 198
- maximum likelihood
 - Dirichlet distribution, 110–111
- MCMC, 167–173, 176
- mean shift algorithm, 138
- median finding, [13](#), 117
- Miller–Rabin test, 32–34
- minimax estimation, 72
- minimum spanning tree, 26–28
- minorization
 - definition, 105
- Minty and Klee example, 78
- MM algorithms, 105
 - ABO, 117
 - advantages, 105
 - ANOVA, 111–113
 - approximate, 110–111
 - closure properties, 105
 - cosine minimization, 109
 - definition, 105
 - Dirichlet, 110–111
 - gamma-Poisson, 118
 - median, 117
 - negative binomial, 109–110
 - nonnegative matrix factorization, 133
 - positron tomography, 113–116
 - power series distribution, 118
 - problems, 116–120
 - quantile, 117
 - Weibull, 119
- modified Gram–Schmidt, 46
- modular arithmetic
 - exponentiation, 31
 - Gauss's theorem on order, 184
 - order of a unit, 183
 - rules, [30](#), 181
 - square roots, 32
 - units, [30](#), 32, 183–184
- Monte Carlo, 159–174
 - random deviates, 160–166
 - randomized matrix multiplication, 166–167
- Moore–Penrose inverse, 198
- multinomial distribution, 191
- multiplicative random number generator, 159–160
- naive Bayes, 126–128, 138
- Nash's method, 100
- negative binomial distribution, 109–110
- Newton's method, 53–70, 108
 - convergence, 54, 62, 68
 - cosine minimization, 109
 - division, 55
 - Lambert's equation, 67
 - least squares, 61
 - logarithms, 56
 - matrix inversion, 57
 - nonlinear least squares, 65–66
 - one-dimensional code, 54
 - optimization, 60
 - quadratic function, 61
 - quasi-Newton, 67
 - random multigraph, 61
 - root extraction, 55, 69
 - step-halving, 63
- nonnegative matrix factorization, 133–137
 - Kullback–Leibler divergence, 139
 - MM algorithm, 133
 - projected gradient algorithm, 135–137
 - shrinkage, 139
- norm
 - ℓ_1 , 87, 195
 - ℓ_∞ , 195
 - Euclidean, xi, 195
 - Frobenius, xi, 69, 129, 134, 139, 140, 195, 196
 - induced matrix, 195
 - spectral, xi, 196
- number theory primer, 180–184

- objective function, 189
- online means and variances, [10](#)
- order relations, 179–180
- ordinary differential equations, 85
- orthogonal matrix, 51, 102, 196

- PageRank algorithm, 88
- Pareto distribution, 175
- partial pivoting, 39
- path-following, 83
- peasant multiplication, [1](#)
- penalty
 - ridge, 129
 - roughness, 115
- perfect power, [29](#)
 - algorithm, [30](#)
- periodogram, 153–154
- permutation matrix, 39, 51
- Poisson distribution, 61, 171
- polar method, 162
- polynomial
 - multiplication, 150
 - number of roots, 183
- positive definite matrix, 103–104
- positron tomography, 113–116
- power means, 186
- power method, 86–89
- power series distribution algorithm, 118
- preconditioning, 49
- Prim’s algorithm, 26–28
- prime number, 181
- prime number theorem, [9](#), [29](#), 180
- principal components, 86
- priority queue, [18](#), [24](#), [26](#), 98
- progeny generating function, 146
- projected gradient algorithm, 135–137
- projective transformation, 78
- Pythagorean triple, [9](#)

- QR decomposition, 44–47, 52, 93
- quadratic bound principle, 107, 116
- quadratic formula, 3–4
- quadratic programming, 192
- quantile
 - MM algorithm, 117
- quantum mechanics, 85
- quasi-Newton method, 67
- quickselect, 13–14
- quicksort, 11–13
 - average-case performance, [12](#)
 - median finding, [13](#)
 - promotion process, [11](#)

- random deviates, generating
 - arcsine, 175
 - discrete, 162
 - gamma, 164
 - Gaussian, 162
 - Gumbel, 175
 - logistic, 174
 - Pareto, 175
 - Poisson, 170
 - slash, 175
 - Weibull, 175
 - Zipf, 165
- random multigraph, 61
- random number generation, 159–160
- randomized matrix multiplication, 166–167
- Raphson, Joseph, 53
- Rasch model, 120
- Rayleigh quotient method, 89
- recurrence relations
 - average-case quicksort, [12](#)
- reflection matrix, 102, 196
- regression
 - ℓ_1 , 72
 - ℓ_∞ , 73
 - linear, 43–45, 52
 - logistic, 63
 - negative binomial, 66
 - nonlinear, 65–66
 - weighted, 69
- renewal equation, 150–151
- revised simplex method
 - code, 76–78
 - computational complexity, 78
 - convergence, 76
 - derivation, 74–76
- ridge penalty, 129
- root extraction, 55
- Rosenbrock’s function, 117
- rotation matrix, 96–98, 102, 196
- row reduction, 37

- second differential, xi
- secular equation, 57
- segmental function, 156–158
- set indicator, xi
- Sherman–Morrison formula, 62, 69, 193
- Sieve of Eratosthenes, [5](#)
- sift-down, [16](#)
- Simpson, Thomas, 53
- simulated annealing, 173–177
- sine transform, 156
- singular value decomposition, 86, 100, 197–201
- skew-symmetric matrix, 102
- slash distribution, 175
- smoothing, 154
- sparse array, 76
- spectral clustering, 86
- spectral decomposition, 194
- spectral density, 153
- spectral norm, xi, 196
- splitting entry, [11](#)
- steepest descent, 78
- step-halving, 63
- Stiefel, Eduard, 47

strong pseudoprime, 33
Sudoku puzzle, 177
supporting hyperplane inequality, 186

theorem

- binomial, 69
- central limit, 164
- Chinese remainder, 33, 34, 182
- contraction mapping, 87, 201
- dominated convergence, 154
- Eckart–Young, 139, 199
- ergodic, 168, 176
- Euler’s, 32
- Fermat’s little, 31, 32, 35, 160, 183
- intermediate value, [16](#)
- Karush–Kuhn–Tucker, 189
- Lagrange’s, [30](#), [34](#)
- prime number, [9](#), [29](#), 180

time series, 153–154

- spectral density, 153

Toeplitz matrix, 152

transportation problem, 72

traveling salesman problem, 173–174

triangle inequality, [24](#), 195

tridiagonal matrix, 51

twin prime conjecture, [29](#), 35

unit simplex, 80

upper triangular matrix, 38, 51, 102

vertex cover problem, 74

Walsh–Hadamard transform, 153

wavelet transform, 153

Weibull distribution, 119, 175

weighted graph, [21](#)

Williams, John, [14](#)

Woodbury’s formula, 69

worst case analysis, [16](#), [18](#)