Daniel Heck

PROGRAMMING IDEAS

Programming Ideas

Daniel Heck

Programming Ideas

http://programmingideas.net

First Edition

Revision 1 (July 1, 2026)

https://this-is-programming.com

Contact: mail@dheck.net

Contents

Preface vii

1 Iterated Function Systems 1
- 1.1 The Chaos Game 2
  - *Implementing the Chaos Game* 4
- 1.2 Generalizing the Chaos Game 7
  - *The Barnsley Fern* 8
  - *Affine Iterated Function Systems* 10
  - *The Extended Chaos Game* 13
- 1.3 Problems 16
  - *Variations of the Chaos Game* 17
  - *Blending Iterated Function Systems* 19
- 1.4 Chapter Notes 21

2 The Mandelbrot Set 23
- 2.1 Computing the Mandelbrot Set 24
  - *The Mandelbrot Iteration* 25
- 2.2 Drawing the Mandelbrot Set 30
- 2.3 The Buddhabrot 35
- 2.4 Problems 39
  - *Drawing and Animating Julia Sets* 39
  - *The Game of Life* 40
- 2.5 Chapter Notes 42

3 Growing Trees 45
- 3.1 Iteration and Recursion 45
- 3.2 Pythagoras Trees 48
  - *Geometric Transformations* 51
  - *Combining Affine Transformations* 53

3.3 Constructing the Pythagoras Tree 56
Transforming Each Square Just Once 57
3.4 Problems 59
Recursive Plants 59
Fibonacci Numbers and Trees 59
The Towers of Hanoi 61
3.5 Chapter Notes 62

4 Recursive Curves 65
4.1 Hilbert Curves 65
Drawing Hilbert Curves 68
4.2 Bézier Curves 70
4.3 De Casteljau's Algorithm 74
Adaptive Subdivision 75
4.4 Problems 78
The Koch Snowflake 78
Closest Points on Lines and Bézier Curves 79
Bézier Curves Continued 80
Penrose Tilings 81
Plotting Parametric Curves 84
4.5 Chapter Notes 86

5 Bit Basics 89
5.1 Storing and Reading Bits 89
5.2 Numbers and Binary Digits 95
Two's-Complement Representation 95
5.3 Computing with Powers of 2 100
Dividing by Powers of 2 101
Trailing Zeros 102
Leading Zeros and Logarithms 103
5.4 Bit Sets 107
Constructing Bit Sets 107
Set Operations 109
5.5 Iterating over Bits 112
Bit Iterators 113
5.6 Problems 116
Cellular Automata 116
Large Bit Sets 117
Counting Bits 119
5.7 Chapter Notes 122

6 Exploring Peg Solitaire **125**
6.1 Representing Boards 126
Bitboards 127
Computing Possible Moves 129
6.2 Backtracking 133
6.3 Counting Solutions 137
Normalization 137
Memoization 140
6.4 Problems 143
Nim 143
Solving Sudoku 145
6.5 Chapter Notes 146

7 Traversing Graphs **149**
7.1 The Die-Hard Graph 151
7.2 Breadth-First Search 156
The Anatomy of Graphs 160
7.3 Retracing Paths 164
7.4 Problems 168
River-Crossing Problems 168
Measuring with Three Jugs 169
Hanoi Graphs 169
Flood Fill 171
7.5 Chapter Notes 172

8 Sokoban and Path Finding **175**
8.1 Boards and Levels 177
Storing Levels 180
8.2 Moving the Player 182
8.3 Pushing Crates 186
8.4 Solving Sokoban 190
Retracing the Worker's Steps 191
8.5 Minimizing the Number of Steps 193
8.6 Dijkstra's Algorithm 197
Implementing Dijkstra's Algorithm 200
Solving Sokoban using the Fewest Steps 202
8.7 Problems 204
Word Ladders 204
Lasers and Mirrrors 205
Path Finding 207
8.8 Chapter Notes 208

9 Encoding Information 211
9.1 Binary Codes 212
Prefix Codes 214
9.2 Implementing Prefix Codes 217
Encoding and Decoding with Prefix Codes 219
Encoding the Code Itself 221
9.3 Huffman Coding 223
9.4 Implementing Huffman Codes 226
Compressing Files using Huffman Codes 228
9.5 Problems 230
Shannon-Fano Codes 230
Bit Streams 231
Unicode and the UTF-8 Encoding 232
Binary Coded Decimals 234
9.6 Chapter Notes 235

10 Lempel-Ziv Compression 237
10.1 The LZ77 Algorithm 238
Compression 239
Decompression 241
10.2 From Bytes to Tokens 244
Cyclic Buffers 246
Producing the Next Token 248
10.3 Finding Matches Quickly 251
Indexing the Dictionary Region 251
Implementing the Index 254
10.4 The LZSS Encoding 256
10.5 Compressing LZ77 Tokens 259
Variable-Length Codes for Integers 261
The LZH Encoding 263
10.6 Comparison of Algorithms 268
10.7 Problems 270
10.8 Chapter Notes 272

11 Big Integers 275
11.1 From Digits to Numbers 276
11.2 Comparison and Conversion 281
11.3 Addition and Subtraction 283
Subtraction 286
11.4 Multiplication 288
11.5 Karatsuba's Algorithm 294
Implementation of Karatsuba's Algorithm 296

11.6 Division and Remainders 298
Finding Digits using Trial and Error 301
Implementing Long Division 305
11.7 Radix Conversion 308
11.8 The Sign-Magnitude Representation 314
Signed Arithmetic 318
11.9 Problems 320
Analysis of Karatsuba Multiplication 320
Integer Powers 320
Decimal Digits 322
Durations 323
Computing Pi 324
11.10 Chapter Notes 327

A Index to Notations 331

B Java Distilled 333
B.1 Classes and Packages 333
B.2 Primitive Types 335
B.3 Data Classes 338
Records 339
B.4 Interfaces 340
B.5 Generics and Collections 342
Collections 343
Iterators and Iterables 344
B.6 Functional Programming 345
Lambda Expressions 347
Side Effects 348
B.7 Predefined Functional Interfaces 349

C Answers to Exercises 353
Preface 353
Chapter 1 353
Chapter 2 359
Chapter 3 364
Chapter 4 370
Chapter 5 379
Chapter 6 392
Chapter 7 401
Chapter 8 416
Chapter 9 428
Chapter 10 438
Chapter 11 446

Bibliography 461

Index of Identifiers 467

Index 481

Preface

Quite exciting, this computer magic!
— Viv Savage, *This is Spinal Tap*

One man's "magic" is another man's engineering.
— Robert A. Heinlein

This is a book about turning your ideas into working programs. It explains how to model problems from a wide range of applications, how to design appropriate data structures, how to implement the required algorithms, and how to combine everything into efficient computer programs. We start with a few simple problems related to the visualization of fractals, continue with programs that are able to solve classical puzzles such as Sudoku, peg solitaire, or Sokoban, and finally discuss how programs like ZIP are able to compress files to a fraction of their original size and how computer algebra systems handle arbitrarily large integers. For each of these problems we will develop complete, working programs and discuss their implementations.

My main goal was to write the kind of book I would have loved to read when I was learning programming back in high school, a book that goes beyond the toy problems discussed in ordinary programming books but doesn't require the mathematical sophistication (or access to teaching assistants) that is assumed by many university textbooks. Here are a few potential target audiences and how they might benefit from reading this book:

- If you are a *high-school student* or a *self-taught programmer*, this book will teach you a wide range of programming techniques and important concepts from computer science. You may find the first few chapters relatively easy, but some of the topics discussed in later chapters will put your programming skills

to the test: it's intentionally a book that grows with you as you become a better programmer.

- If you are an *undergraduate student*, this books can serve as a complement to more rigorous courses on programming and algorithms and will teach you how to apply the concepts taught in class. If your courses make you think that programming is dry and abstract, read on for practical applications that are both fun and instructive.
- And finally, as an *expert programmer* or *professional computer scientist*, you will probably appreciate the many programming problems and computational puzzles collected in this book. You will also find that reimplementing the example programs is a great way to learn and evaluate new programming languages.

Ideally, before reading this book, you should already know how to program and how to compile and run your code. In addition, we will assume that you are familiar with fundamental concepts such as recursion, arrays, linked lists, and hash tables. A high school level of mathematics is sufficient to understand most of the material, but we will make occasional use of more advanced topics such as linear algebra, induction, big-Oh notation, or computational complexity; See Appendix A for an overview of some of the notations used in this book.

The programming language we will use to write all example programs is *Java*, mainly because it is widely used and taught and strikes a good balance between high-level conveniences and low-level features. If you are coming from another language, you will find a summary of Java's main features in Appendix B.

The book's official home page can be found at

`https://programmingideas.net`

In addition, the book's GitHub page at

`https://github.com/dheck/programmingideas`

hosts the source code of all example programs and serves as the official bug tracker. Reports of any errors in the text or the examples are greatly appreciated!

Organization

Thematically, the book consists of the following five parts.

Iteration In the first two chapters, we explore one of the simplest yet most powerful ideas in computer science: *iteration*, that is, performing a certain sequence of operations repeatedly. To illustrate different aspects of iteration, we will mainly study ***fractals*** — mathematical objects that have detail at every level of magnification and that are almost impossible to visualize without the use of computers. In Chapter 1 we will discuss ***iterated function systems***, which are procedures for generating geometric shapes that consist of countless smaller copies of themselves. To visualize these shapes, we employ the ***chaos game***, a simple algorithm that demonstrates how even simple loops pack a surprising amount of computational punch.

In Chapter 2 we turn to another famous fractal called the ***Mandelbrot set***. To visualize the Mandelbrot set, we will need to learn about complex numbers and how to use iteration to evaluate infinite sequences and compute the pixels of raster images. We also discuss a variation of the Mandelbrot set called the ***Buddhabrot***.

Recursion In the next two chapters we turn to recursion, an alternative way of expressing repetition in computer programs. Unlike iteration, which regards computations as long, linear sequences of operations, recursion decomposes them into a tree-shaped hierarchy of smaller and smaller computations. In Chapter 3 we explain how to use recursion to construct a fractal known as the ***Pythagoras tree***.

Chapter 4 discusses two other geometric objects that can be defined recursively: ***Hilbert curves*** and ***Bézier curves***. The former are an example of a so-called *space-filling curve*, a curve that is coiled up so tightly that it ultimately converges toward a solid object in the plane. The Bézier curves covered in the second half of the chapter are widely used in computer graphics; among other things, we they are used to describe the smooth curves in typefaces and vector graphics. Bézier curves have an unexpected recursive structure, which is the basis of several elegant algorithms for working with them.

Bits We then turn our attention to ***bits***, the fundamental unit of storage (and computation) inside a computer. We start with an overview of programming techniques for working with bits in Chapter 5. We will discuss how to read and modify individual bits and groups of bits, learn about ***two's complement representation*** for storing signed integers, and develop a wide range of useful techniques for performing computations that involve powers of 2 and binary logarithms.

In Chapter 6 we then use some of these bit-processing techniques to study a classic puzzle called ***peg solitaire***. We will see how to use bits to represent the possible states of this puzzle and how to use bit operations to generate all possible moves and solutions. We will also study ***backtracking***, a simple, recursive method for systematically generating chains of moves (or arbitrary other objects), as well as techniques for speeding it up.

Graph traversal We continue with games and puzzles in the next two chapters, but this time the focus is on problems that can be modeled and solved using ***graphs*** and ***graph algorithms***. As we shall discuss, graphs are a general model for describing and analyzing the relationships between various kinds of objects — including the relationships between the possible states of many games and puzzles.

In Chapter 7, we discuss the basics of graph theory, how to model simple puzzles in this framework, and how to solve them algorithmically using elementary techniques such as breadth-first search. In Chapter 8, we tackle a significantly more challenging game and study how to find optimal solutions for the classic computer game ***Sokoban***. Our investigation of Sokoban will naturally lead us to the notion of a ***weighted graph*** and ***Dijkstra's algorithm***, an important generalization of breadth-first search.

Data compression The next two chapters focus on data compression, the art and science of reducing the number of bits needed to store or transmit information. We start in Chapter 9 with an overview of ***binary codes***. A binary code can be thought of as a procedure for translating a sequence of objects (for example, the bytes in a file) to a string of bits. The main question is: How can we design binary codes that use as few bits as possible but still allow us to recover the original sequence of objects? We will discuss an important class of codes known as ***prefix codes*** and a fundamental technique for constructing efficient prefix codes known as ***Huffman coding***.

In Chapter 10 we discuss the implementation of ***LZ77***, a general-purpose compression algorithm that is the basis of popular file formats such as `zip` and `png`. From an algorithmic perspective, LZ77 is fairly simple, but its implementation is far from trivial. We will discuss how to make the algorithm run quickly, how to design an efficient bit encoding for the output, and how to combine LZ77 with Huffman coding to achieve even better compression rates.

Arithmetic We conclude in Chapter 11 with an in-depth discussion of ***big integers***, a data type that allows us to compute with arbitrarily large integers. Conceptually, big integers are closely related to the familiar decimal numbers, and the standard arithmetic procedures for adding, multiplying, and dividing decimal numbers have natural generalizations to big integers. There are several crucial differences, however. On the one hand, we must contend with computer-specific challenges such as memory management and integer overflow. On the other hand, the use of computers allow us improve some of the algorithms in unique ways, for example by exploiting certain arithmetic tricks or by using algorithmic techniques that are hard to carry out by hand.

Code Examples

Object-oriented programming languages like Java organize complex programs by dividing them into multiple classes with well-defined responsibilities. Each class consists of fields that hold some of the program's state and methods that operate on this state and perform some of the program's computations. Classes can be nested inside other classes, and the top-level classes are organized into packages, which can be nested inside other packages.

This kind of hierarchical organization makes it possible to divide large programs into manageable, self-contained units, but it's not a good match for presenting

code in a book. We will therefore split complex classes and their nested parts into multiple smaller listings, which we then discuss separately. Listing 0.1 illustrates how this would look for a class `Hello` with one field `greeting` and some additional code that isn't shown. The label before the listing tells us that the code is part of a class called `Hello` that is defined in a package called `preface`. In general, the source code for every chapter can be found in a separate package; for example, the classes in Chapter 1 are all defined in a package called `ch1`.

Listing 0.1 preface / Hello

```
1  public class Hello {
2      private String greeting;
3
4      // ...
5  }
```

The comment '`// ...`' on line 4 of Listing 0.1 indicates that some nested parts of the class have been omitted. In this case, we have moved the implementation of a method called `printGreeting()` to Listing 0.2. The label above the listing tells us that the code also belongs to the class `preface/Hello`.

Listing 0.2 preface / Hello

```
1  public void printGreeting() {
2      System.out.println(greeting);
3  }
```

To make it easier to navigate the many code examples, most pages include a small index of referenced names in the margin area. Here is an example of such an index:

```
Hello 0.1
  greeting 0.1
  printGreeting() 0.2
String →
System →
```

Each line tells us where a certain name is defined. In this case, `Hello` and its member `greeting` are defined in Listing 0.1, while the method `printGreeting()` is defined in Listing 0.2. (In the PDF version of the book, you can jump to the definition by clicking on the listing number.) If an identifier is defined on the current page, its entry is shown in bold. Classes defined in the standard library are shown in italics, and clicking the arrow symbol (→) next to the name will take you to the respective page in Java's online documentation.

In addition to these per-page indexes, there is a more extensive "Index of Identifiers" at the end of this book that lets you quickly find the definitions and major uses of all classes and methods defined in the book.

Hello 0.1
greeting 0.1
printGreeting() 0.2
String →
System →

Exercises and Problems

More than 130 exercises are included in the book, most of which are easy and primarily designed to review or reinforce the material presented in each section. Exercises with a mathematical flavor are marked with a star (⋆).

Exercise 0.1. How many Java programmers does it take to change a light bulb?

⋆ **Exercise 0.2.** Assume that one programmer can change a light bulb in one minute. How long does it take n programmers to change n light bulbs?

In addition, every chapter ends with a selection of *problems*, which are larger exercises that expand on the material covered in the chapter in nontrivial ways. Some of this book's best material can be found in these problems, so be sure to give them a try! Complete answers to most exercises and problems are provided in Appendix C.

Chapter 1

Iterated Function Systems

Insanity is doing the same thing
over and over again
and expecting different results.
— Anonymous[1]

Modern computers are remarkably powerful devices, capable of storing millions of pages of text and executing tens of billions of instructions per second, all while being small enough to fit in the palm of your hand.

The key to harnessing all this power is ***repetition***, the seemingly simple idea of performing a sequence of operations multiple times. As a simple example, suppose we want to print a million integers starting from zero. Every programmer knows how to do this: Use a loop to count from 1 to 1 000 000 and output the successive values of the loop variable `i`:

```
for (int i = 1; i <= 1_000_000; i++) {
    System.out.println(i);
}
```

This form of repetition is known as ***iteration***, and the example demonstrates its two main ingredients: Every iterative procedure repeatedly performs a sequence of operations (here the instructions inside the loop) while updating a set of variables with each repetition (here the loop variable `i`). It's the ability to update variables that distinguishes iteration from mindless repetition and allows us to do *slightly different things* over and over again.

[1] The quote is often mistakenly attributed to Albert Einstein; see [79].

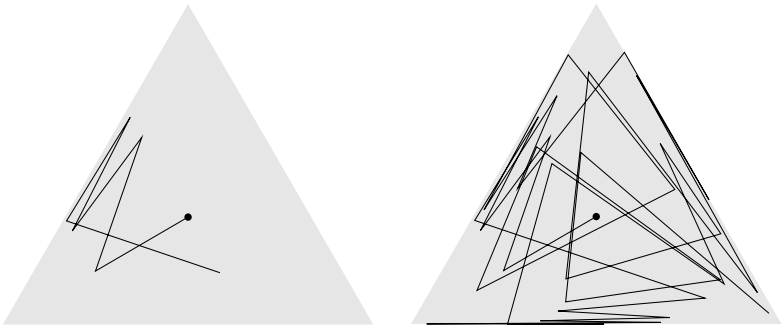


Figure 1.1 The first steps of the chaos game. The game starts with a single point in the center of the triangle and then repeatedly moves it halfway towards one of the corners chosen at random. The image on the left shows the first six points generated in one particular run of this game, and the image on the right the first 50 steps.

1.1 The Chaos Game

Iteration is so commonplace that we rarely give it the recognition it deserves — what could be more ordinary than a loop that iterates over a sequence of integers or the elements of a list? But it's not hard to find examples of simple iterative processes that show startlingly complex behavior. One such algorithm is the ***chaos game***, a simple procedure for constructing a sequence of points in the plane.

The chaos game simulates the movement of a point in the plane. We start with a point somewhere inside an equilateral triangle, for example in its center, and then repeatedly move it halfway toward a randomly chosen corner. The trajectory of the point depends on the order in which the corners are chosen. Figure 1.1 shows the first steps from one particular run of the chaos game. In the left part of the image, the point starts its journey by moving alternately towards the lower-left and upper corners and then takes a final step towards the lower-right corner. Afterward, as shown in the right part of the image, the point continues to move erratically inside the triangle.

You might expect that the points generated by the chaos game will eventually cover the entire triangle more or less evenly. This is not what happens, however. Figure 1.2 shows the point distributions after 50 000 and 200 000 iterations. As you can see, the points produced by the chaos game are distributed unevenly and converge toward a shape that looks like a triangle from which a number of successively smaller triangles have been cut out. There is a single large empty triangle in the center, which is surrounded by three smaller ones. Each of these smaller triangles is again surrounded by three smaller triangles, and the pattern continues indefinitely. The longer we let the chaos game run, the more closely the resulting set of points approximates this unexpected limiting shape.

This curious shape — known as the ***Sierpiński triangle*** — is an example of a ***fractal***. Fractals are geometric objects that share two main properties: They have a "fractured" appearance that is irregular but not random, and they can be magnified

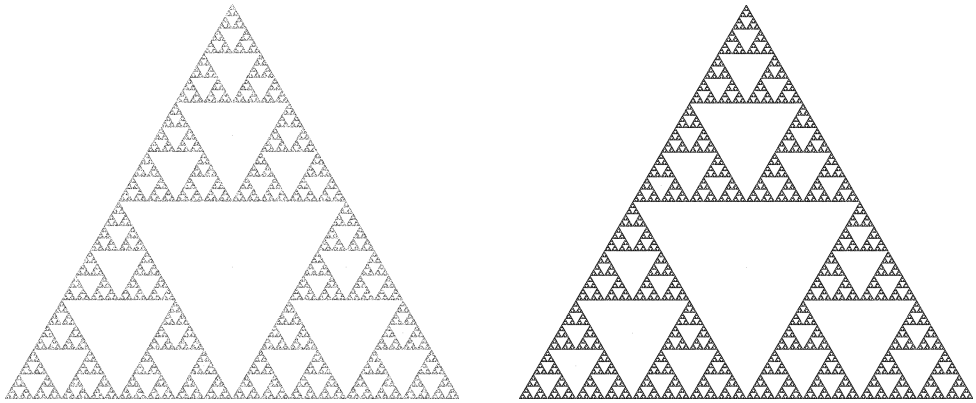


Figure 1.2 Visualization of the first 50 000 and 200 2000 points computed by the chaos game. As the number of iteration increases, the point set slowly converges toward a fractal called the Sierpiński triangle.

indefinitely, revealing new details at every level of magnification. In addition, fractals are often ***self-similar***, which means that their parts resemble the shape as a whole. The Sierpiński triangle clearly has all three properties: Its geometric structure isn't random but it's also not as regular as a simple polygon or circle, and it can be magnified indefinitely, with new copies of the Sierpiński triangle appearing at every level of magnification.

The chaos game we used to construct the Sierpiński triangle is a simple example of an ***algorithm***, a formal description of how to break down a computation into a sequence of elementary steps. To describe algorithms, we will use the format shown in Algorithm 1.1. In this case, the algorithm consists of a single function named ChaosGame that takes two inputs $\boldsymbol{s}$ and n, the starting point and the number of iterations to perform. The numbered lines that follow explain the computational steps performed by the algorithm.

Algorithm 1.1

Given a starting point $\boldsymbol{s}$, produce a sequence of n points in the plane.

ChaosGame($\boldsymbol{s}, n$) =
1 **let** $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ = corners of an equilateral triangle
2 **var** $\boldsymbol{p} = \boldsymbol{s}$
3 **repeat** n times
4 **output** $\boldsymbol{p}$
5 **let** k = random integer between 1 and 3
6 $\boldsymbol{p} = (\boldsymbol{p} + \boldsymbol{c}_k)/2$

Let's take a closer look at the steps of Algorithm 1.1. Line 1 defines three variables $\boldsymbol{c}_1$ to $\boldsymbol{c}_3$ that represent the corners of an equilateral triangle; we assume that these corners are already known or that they are computed using a method

not specified in the algorithm. We use the **let** keyword to define read-only variables that don't change over the course of the algorithm. Line 2 defines a new variable $\boldsymbol{p}$ for the current position and initializes it to the starting point $\boldsymbol{s}$. This variable is defined using the **var** keyword, which indicates that the value of $\boldsymbol{p}$ is modified as the algorithm progresses. The **repeat** loop on the following line repeats the indented lines in its body n times. It starts with an **output** statement that outputs the current value of $\boldsymbol{p}$. Line 5 then sets the constant k to a random integer between 1 and 3, and line 6 moves $\boldsymbol{p}$ toward the kth corner. The loop is then repeated $n-1$ more times. Since **output** is executed n times, the algorithm produces n points.

> Mathematicians usually distinguish between ***points*** and ***vectors***: Points represent *locations* in space, whereas a vectors represent *directions*. However, for simple geometric problems in the plane, the two concepts are closely related: Every point P with coordinates (x, y) can be represented by the ***position vector*** $\boldsymbol{p} = \binom{x}{y}$ that points from the origin to P, and vice versa.
>
> For the geometric problems discussed in this book, we will keep things simple and use points and positions vectors interchangeably. Boldface symbols like $\boldsymbol{p}$ and $\boldsymbol{q}$ are always vectors, and if we say something like "the point $\boldsymbol{s}$," what we really mean is "the position vector $\boldsymbol{s}$."

Implementing the Chaos Game

We can now turn to the fun part: implementing the chaos game so that it runs on a real computer and produces real images. In most cases, turning an algorithm into a working program is more complicated than translating it step by step. For instance, we may have to make changes based on the chosen programming language, flesh out details that weren't specified precisely, or integrate the algorithm with other parts of a larger program. Here are some of the technical details we have to consider when implementing the ChaosGame algorithm:

1. How do we store points and move them?
2. How do we compute the three corners of the outer triangle?
3. How do we create random numbers between 1 and 3?
4. What does it mean to **output** a point?

Let's address these questions one by one.

We start by defining a custom data type called `Point` that allows us to store and compute with points in the plane. What kinds of operations do we require in `Point`? Obviously, we need a way to construct points at certain positions in the plane, so that we can specify the starting point of the chaos game and the three corners that enclose the fractal. To store a point's position we use ***Cartesian coordinates*** (x, y), which measure the point's position along the horizontal and vertical axis. The `Point` type defined in Listing 1.1 stores these coordinates in two fields `x` and `y`.

`Point` **1.1**

Listing 1.1 ch1/Point

```
1  // A point in 2D.
2  public record Point(double x, double y) {
3      // ...
4  }
```

To implement the chaos game, we also need a way to move the current point halfway toward one of the triangle's corners, that is, we need to compute the ***midpoint*** of two points. The easiest way to do this is to average their coordinates, as shown here for the points p and q:

```
Point midPoint = new Point(
        0.5 * (p.x() + q.x()), 0.5 * (p.y() + q.y()));
```

This is a special case of a more general operation that computes points along the line segment between two arbitrary points $\boldsymbol{p}$ and $\boldsymbol{q}$. In computer graphics, the point that lies some fraction t between $\boldsymbol{q}$ and $\boldsymbol{q}$ is commonly denoted by $\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t)$.[2] Three special cases of this ***lerp function*** are worth noting: For $t = 0$ we obtain the first end point $\boldsymbol{p}$, for $t = 1$ the second end point $\boldsymbol{q}$, and for $t = 1/2$ their midpoint. The function is illustrated in Fig. 1.3.

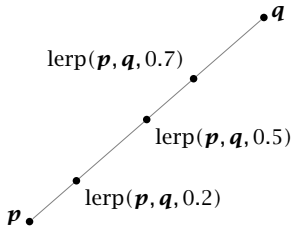


Figure 1.3 The lerp function computes a point that lies on the line segment between two end points.

There are two equivalent formulas for defining the lerp function. The first is obtained by starting at the first point $\boldsymbol{p}$ and then advancing along the line to $\boldsymbol{q}$ by adding the difference vector $\boldsymbol{q} - \boldsymbol{p}$ scaled by t:

$$\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t) = \boldsymbol{p} + t(\boldsymbol{q} - \boldsymbol{p}).$$

The second formula has a more algebraic flavor and expresses the lerp function as the weighted average of $\boldsymbol{p}$ and $\boldsymbol{q}$:

$$\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t) = (1 - t)\boldsymbol{p} + t\boldsymbol{q}. \tag{1.1}$$

Both definitions can be transformed into each other by reorganizing the terms on the right-hand side. Even though the first definition is slightly more intuitive, we

Point 1.1
Point() 1.1
x() 1.1
y() 1.1

[2] The name "lerp" is a contraction of "linear interpolation."

will use the second one since it is more symmetrical and slightly easier to generalize to other forms of interpolation; the implementation is shown in Listing 1.2.

Listing 1.2 ch1/Point

```
// Compute the linear interpolation of two points.
public static Point lerp(Point p, Point q, double fraction) {
    return new Point((1 - fraction) * p.x + fraction * q.x,
            (1 - fraction) * p.y + fraction * q.y);
}
```

It's now easy to implement the chaos game itself. The `chaosGame()` function in Listing 1.3 takes three arguments — a list of corners, a starting point, and the number of iterations that should be performed — and returns the sequence of points it visits as a list of `Points`. The function first creates a random number generator `rng` that is used to select one of the corners in each iteration and then allocates a data structure for the return value, in this case an `ArrayList` of `Points`.[3] The actual algorithm is implemented by the `for` loop that follows, which stores the current point in `points` and then moves it towards a randomly chosen corner.

Listing 1.3 ch1/ChaosGame

```
public static List<Point> chaosGame(List<Point> corners,
        Point start, int iterations) {
    var rng = new Random();
    var points = new ArrayList<Point>(iterations);
    Point p = start;
    for (int i = 0; i < iterations; i++) {
        points.add(p);
        Point c = corners.get(rng.nextInt(corners.size()));
        p = Point.lerp(p, c, 0.5);
    }
    return points;
}
```

If we want to create an image of the Sierpiński triangle like the one in Fig. 1.2, we have to simulate a few thousand iterations of the chaos game and then draw the resulting points. One way to do this is shown in Listing 1.4. We first construct a large equilateral triangle with a side length of 1000 and then call `chaosGame()` to compute 50 000 points inside this triangle, using the center of the triangle as the starting point. Finally, we draw each of the resulting points as a small, filled circle. The `Graphics2D` type used in this listing is the standard interface provided by Java for producing 2D graphics; Exercise 1.7 on page 19 asks you to complete this program by setting up a suitable instance of `Graphics2D`.

ArrayList →
ChaosGame
  chaosGame() 1.3
  drawSierpinski() **1.4**
Graphics2D →
Point **1.1**
  Point() **1.1**
  lerp() 1.2
Random →

You may be wondering why `drawSierpinski()` removes the first 50 elements of `points` before drawing them. The reason is that for some starting points the

[3] In Java, an `ArrayList` is a special type of `List`, so the code is correct even though the return type of `chaosGame()` doesn't match the type of `points` exactly.

Listing 1.4 ch1/ChaosGame

```
public static void drawSierpinski(Graphics2D g) {
    double sideLength = 1000;
    var corners = List.of(
            new Point(0, 0),
            new Point(sideLength, 0),
            new Point(sideLength / 2, sideLength * Math.sqrt(3) / 2));
    Point start = new Point(
            sideLength / 2, sideLength / Math.sqrt(3) / 2);
    List<Point> points = chaosGame(corners, start, 50_000);

    float radius = 0.01f;
    for (Point p : points.subList(50, points.size()))
        g.fill(new Ellipse2D.Double(p.x(), p.y(), radius, radius));
}
```

chaos game may require a few iterations to "warm up" and produce points that are sufficiently close to the actual fractal. In the present example, for instance, we start at the center of the triangle, which isn't part of the Sierpiński triangle. But with every iteration, the point moves closer to the final fractal until it is imperceptibly close. Discarding the first few points produced by the chaos game is enough to get rid of the obvious outliers and obtain clean-looking images.

Exercises

Exercise 1.1. Given two points $\boldsymbol{p}$ and $\boldsymbol{q}$, what is the geometric interpretation of $\operatorname{lerp}(\boldsymbol{p}, \boldsymbol{q}, -1)$?

Exercise 1.2. Extend the `Point` class with a method `distance()` that computes the distance between two points.

Exercise 1.3. What happens if you place the starting point of the chaos game outside of the triangle defined by the three corners? What happens if you move the corners of the triangle so that it is no longer equilateral?

1.2 Generalizing the Chaos Game

The chaos game as we have discussed it in the previous section can be generalized to a more powerful algorithm for generating fractals. As before, we simulate the movement of a point in the plane, but instead of moving the point toward a randomly chosen corner of a triangle, we instead transform it using a randomly chosen function. For reasons we will explain in a moment, we usually associate with each function f_k a probability p_k, which determines how frequently the chaos game chooses this function relative to the others. We call the combination of functions and their probabilities an ***iterated function system*** or ***IFS***.

ChaosGame
  chaosGame() **1.3**
  drawSierpinski() 1.4
Ellipse2D →
Graphics2D →
List →
Math →
Point **1.1**
  Point() **1.1**

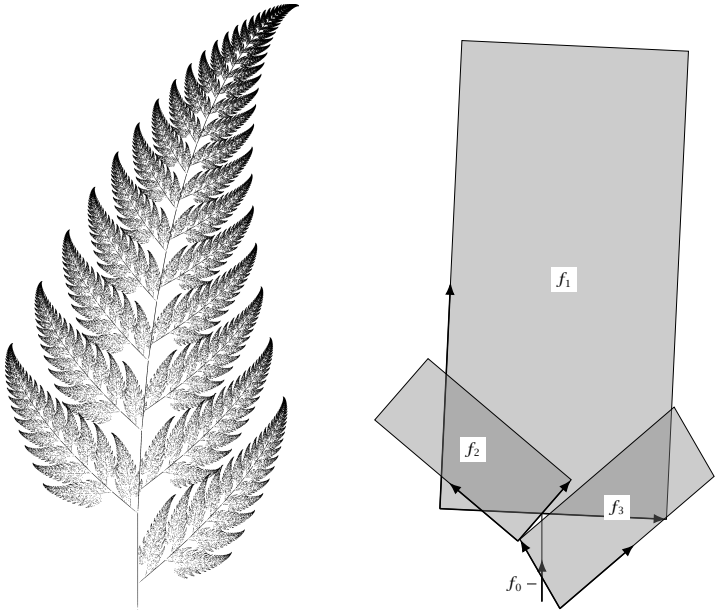


Figure 1.4 The Barnsley fern is a plant-like fractal that consists of countless smaller copies of itself (left). It is generated by four affine transformations, each of which corresponds to one part of the fractal (right).

It's easy to see that this is a proper generalization: If $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ are the corners of a triangle, we can define three functions that move a given point $\boldsymbol{p}$ halfway toward each of the corners:

$$f_1(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_1)/2, \qquad f_2(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_2)/2, \qquad f_3(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_3)/2.$$

If we start with a point inside the triangle and repeatedly transform it using a randomly chosen function from the set $\{f_1, f_2, f_3\}$, we end up with a random process that is equivalent to the chaos game from the previous section. In the case of the Sierpiński triangle, all three corners are chosen with the same probability, so we set $p_k = 1/3$. When visualizing fractals that are less symmetric, these probabilities can be used to control the distribution of the points produced by the chaos game.

The Barnsley Fern

A famous example of a fractal based on iterated function systems is the ***Barnsley fern*** shown in Fig. 1.4. The fern consists of a stem that bends to the right and an infinite number of branches and leaves that are attached to the left and the right of the stem. Each branch is itself a scaled and rotated version of the Barnsley fern, as are the leaves of each branch: If you zoom into any part of the Barnsley fern, you find an endless number of smaller copies of the whole shape.

The Barnsley fern is generated by an iterated function system that consists of four functions and four probabilities:

$$f_0(\boldsymbol{p}) = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad p_0 = 0.01$$

$$f_1(\boldsymbol{p}) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix} \qquad p_1 = 0.85$$

$$f_2(\boldsymbol{p}) = \begin{pmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix} \qquad p_2 = 0.07$$

$$f_3(\boldsymbol{p}) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 0.44 \end{pmatrix} \qquad p_3 = 0.07$$

All four functions have the same form: They first multiply the vector $\boldsymbol{p}$ by a 2×2 matrix and then translate the result by adding a vector. Functions of this form are called ***affine transformations***, and they describe a wide range of geometric transformations such as rotation, scaling, shearing, and translation; see also Box 1.1 on the next page.

The right part of Fig. 1.4 illustrates how these four affine transformations determine the shape and structure of the Barnsley fern. Imagine a rectangle that encloses the Barnsley fern and then transform this bounding box using each of the four transformations. The first transformation f_0 flattens the box into a short vertical line that forms the stem of the fern; f_1 moves a slightly scaled and rotated version of the bounding box to the end of the stem and generates the upper part of the fern; and the last two transformations f_2 and f_3 form the first leaf on the left and right of the fern, respectively. Notice that f_3 also flips the coordinate system and introduces a slight shear, so the leaves on the right side of the fern bend to the left and are slightly skewed. Smaller details of the fern are obtained by applying multiple functions in sequence. For instance, applying f_2 followed by f_0 produces the stem of the left branch, and applying f_1 followed by f_3 the second branch on the right. Every part of the fractal corresponds to a particular combination of the four functions in its iterated function system; you can think of the four functions as the fractal's geometric blueprint.

This geometric blueprint is filled with a random distribution of points using the chaos game. The probabilities $p_1, \ldots, p_3$ control *how* the points are distributed over the different parts of the fractal: In the case of the Barnsley fern, the probabilities are 0.01, 0.85, 0.07, and 0.07, which means that approximately 1% of all points will be placed in the fern's stem (or one of its many copies), 85% in its body, and 7% in the left and right leaves each. The four probabilities sum to 1, which reflects the fact that every point is allocated to one of the available transformations.

It is instructive to consider what happens if we choose other values for the probabilities. If we set all probabilities to $1/4$, for instance, each part of the fractal receives the same number of points, and we obtain the rendering of the Barnsley fern shown in the left part of Fig. 1.5. With this choice of probabilities, most points end up in or close to the stem of the Barnsley fern and fewer points in its leaves. If, on the other hand, we set the four probabilities to 0.01, 0.94, 0.025, and 0.025,

Box 1.1: Linear and Affine Transformations

Many important geometric transformations can be described mathematically using matrix operations. For example, scaling vectors by a constant factor α

$$\boldsymbol{p} \quad \mapsto \quad \alpha \boldsymbol{p}$$

can also be written as the matrix product

$$\boldsymbol{p} \quad \mapsto \quad \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \boldsymbol{p},$$

and similar matrix versions exist for other transformations such as *rotation* around the origin, *shearing*, and *reflection*. In general, a mapping that can be described by a matrix multiplication is called a *linear transformation*:

$$\boldsymbol{p} \quad \mapsto \quad \boldsymbol{A}\boldsymbol{p} \qquad \text{or} \qquad \begin{pmatrix} x \\ y \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}.$$

An important generalization of linear transformations are *affine transformations*, which first perform a matrix multiplication and then add a fixed vector $\boldsymbol{t}$:

$$\boldsymbol{p} \quad \mapsto \quad \boldsymbol{A}\boldsymbol{p} + \boldsymbol{t} \qquad \text{or} \qquad \begin{pmatrix} x \\ y \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} ax + by + t_x \\ cx + dy + t_y \end{pmatrix}.$$

Affine transformations let us rotate, scale, and shear as before, but they can also be used to move geometric objects in the plane. We will discuss affine transformations in more detail in Section 3.2.

most points are allocated to the function f_1, which increases the density of points in the tips of the branches, as illustrated in the right part of Fig. 1.5.

In practice, suitable values for the probabilities must usually be determined by trial and error, especially if the primary goal is to produce pleasing images. But it is possible to compute reasonable starting values so that more points end up in parts of the fractal that cover a large surface area; we will discuss one such an approach in Exercise 1.5.

Affine Iterated Function Systems

An affine transformation is a function that consists of a linear transformation — usually a combination of rotation, scaling, and shearing — followed by a translation. We usually write affine transformations in the following form:

$$\boldsymbol{p} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}. \tag{1.2}$$

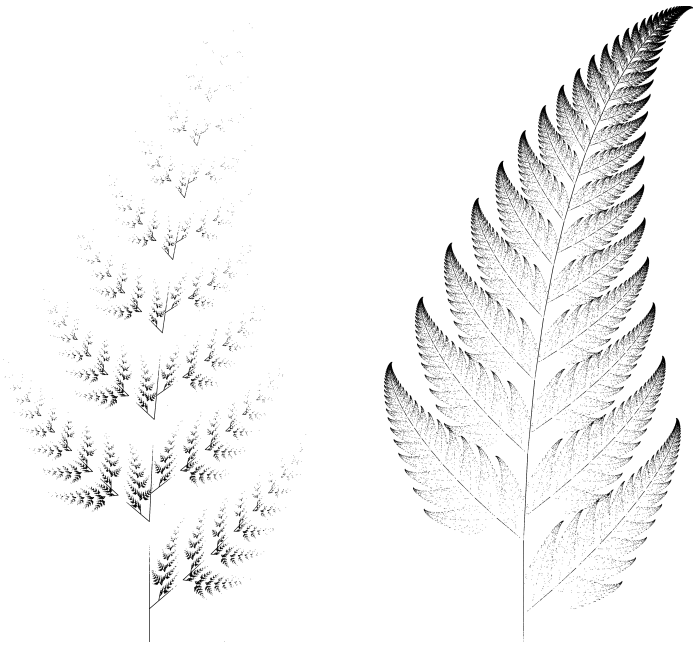


Figure 1.5 The influence of IFS probabilities. The probabilities associated with an iterated function system determine the spatial distribution of points produced by the chaos game. Decreasing the probability of the stem places more points in the leaves (left) whereas increasing it emphasizes the stem but places fewer points in the leaves (right).

where $\boldsymbol{p}$ is the point being transformed. The transform is completely defined by six numbers: the four coefficients a, b, c, d of the 2×2 matrix and the two coefficients t_x, t_y of the translation vector. To implement affine transformations, we define a record called `Affine` that stores these six numbers as floating-point numbers (Listing 1.5).

Listing 1.5 ch1/Affine

```
// An affine transformation in 2D.
public record Affine(
        double a, double b,
        double c, double d,
        double tx, double ty) {
    // ...
}
```

If we set $\boldsymbol{p} = \begin{pmatrix} x \\ y \end{pmatrix}$ and expand the matrix product in Eq. (1.2), we obtain the coordinates x' and y' of the transformed point:

$$\begin{aligned} x' &= ax + by + t_x, \\ y' &= cx + dy + t_y. \end{aligned} \tag{1.3}$$

Affine 1.5
Affine() 1.5

The `transformPoint()` method in Listing 1.6 uses this formula to apply an affine transformation to a given `Point`, producing a new `Point`. Since we will often work with lists of points, we also define a second method `transformPoints()` that transforms multiple points and returns the result as a new list. That's all the functionality we need for now, although we will extend `Affine` with few additional methods in Chapter 3.

Listing 1.6 ch1 / Affine

```
// Transform a single point.
public Point transformPoint(Point p) {
    return new Point(a * p.x() + b * p.y() + tx,
            c * p.x() + d * p.y() + ty);
}

// Transform a list of points.
public List<Point> transformPoints(List<Point> points) {
    var result = new ArrayList<Point>(points.size());
    for (Point p : points)
        result.add(transformPoint(p));
    return result;
}
```

We can now represent an affine iterated function system as an array of affine transformations of type `Affine` together with an array of probabilities of type `double` (Listing 1.7). The constructor of `AffineIFS` initializes `transforms` using the argument of the same name, but for `probabilities` it performs an additional normalization step to ensure that its entries sum to 1. Normalizing the probabilities ensures that exactly one of the affine transformations is chosen by the chaos game in each step. The second argument of `AffineIFS()` is therefore simply an array of *frequencies* that indicate how often each affine transformation is chosen relative to the others, which is then converted into a table of *probabilities* by dividing each entry by `total`, the sum of all frequencies.

Listing 1.8 demonstrates how to construct an instance of `AffineIFS` that represents the Barnsley fern. In this case, the probabilities in the second array are already normalized.

Affine **1.5**
  transformPoint() 1.6
  transformPoints() 1.6
AffineIFS **1.7**
  AffineIFS() **1.7**
  probabilities **1.7**
  transforms **1.7**
ArrayList →
List →
Point **1.1**

Listing 1.7 ch1/AffineIFS

```
1  // Representation of an affine iterated function system.
2  public class AffineIFS {
3      public final Affine[] transforms;
4      public final double[] probabilities;
5
6      public AffineIFS(Affine[] transforms, double[] frequencies) {
7          this.transforms = transforms;
8          this.probabilities = new double[frequencies.length];
9          double total = 0.0;
10         for (double freq : frequencies)
11             total += freq;
12         for (int i = 0; i < frequencies.length; i++)
13             this.probabilities[i] = frequencies[i] / total;
14     }
15
16     // ...
17 }
```

Listing 1.8 ch1/AffineIFS

```
1  public static AffineIFS BARNSLEY_FERN = new AffineIFS(
2          new Affine[]{
3                  new Affine(0, 0, 0, 0.16, 0, 0),
4                  new Affine(0.85, 0.04, -0.04, 0.85, 0, 1.6),
5                  new Affine(0.2, -0.26, 0.23, 0.22, 0, 1.6),
6                  new Affine(-0.15, 0.28, 0.26, 0.24, 0, 0.44),
7          },
8          new double[]{0.01, 0.85, 0.07, 0.07}
9  );
```

The Extended Chaos Game

We can now turn to the problem of implementing the generalized version of the chaos game that lets us visualize the fractals described by affine iterated function systems. The formal description of this algorithm is shown in Algorithm 1.2. The main difference compared to the original chaos game in Algorithm 1.1 is that the algorithm now uses a fixed set of functions $f_1, \ldots, f_K$ and associated probabilities $p_1, \ldots, p_K$ to move the current point in each step. In Line 4, we choose one of the functions by calling a function ChooseIndex, which returns an integer k between 1 and K, and in Line 5, we move the current point $\boldsymbol{p}$ to its new position by transforming it using the kth function f_k.

The job of the ChooseIndex function referenced in Algorithm 1.2 is to take a sequence of probabilities $p_1, \ldots, p_K$ and generate a random index between 1 and K so that the integer k occurs with probability p_k (in other words, it returns the number 1 with probability p_1, the number 2 with probability p_2, and so on). How can we implement ChooseIndex?

Affine 1.5
  Affine() 1.5
AffineIFS 1.7
  AffineIFS() 1.7
  BARNSLEY_FERN 1.8
  probabilities 1.7
  transforms 1.7

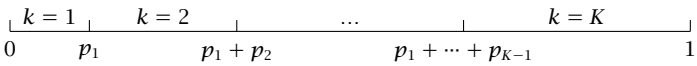
Algorithm 1.2

Given a starting point $\boldsymbol{s}$, produce a sequence of n points in the plane. The variables $f_1, f_2, \ldots f_K$ are affine transformations and $p_1, p_2, \ldots p_K$ the associated probabilities.

EXTENDEDCHAOSGAME($\boldsymbol{s}, n$) =

```
1   var p = s
2   repeat n times
3       output p
4       let k = CHOOSEINDEX(p_1, ..., p_K)
5       p = f_k(p)
```

One solution is to approach the problem geometrically. If the probabilities are normalized, we have $p_1 + \cdots + p_K = 1$, which we can visualize by subdividing the interval $[0, 1]$ into K sub-intervals of length $p_1, p_2, \ldots$:



The kth interval starts at $p_1 + \cdots + p_{k-1}$ and ends at $p_1 + \cdots + p_k$ and therefore has length p_k. If we now pick a random real number u between 0 and 1, the probability that u lands in any particular sub-interval is obviously equal to the length of that interval, and the probability that it lands in the kth interval is p_k. The index of the interval that contains u is therefore an integer between 1 and K with the desired probability distribution.

To find the interval that contains u we can simply try them one after another. If $u < p_1$, the number lies in the leftmost interval with index $k = 1$; if $u < p_0 + p_1$, it lies in the second interval with index $k = 1$; otherwise, we continue in this manner until we find the first index k such that $u < p_0 + p_1 + \cdots + p_k$. The resulting algorithm is shown in Algorithm 1.3. The variable s accumulates the probabilities so it always contains the upper limit of the current interval, and the smallest index for which $u < s$ is the desired random index.

The generalized version of the chaos game can now be implemented as shown in Listing 1.9. The `extendedChaosGame()` method simulates the movement of the point `start` over the specified number of `iterations` and returns a list of visited points. In each iteration, the function appends the current position to `points` and then advances to the next point applying one of the affine transformations chosen at random. The `chooseIndex()` method implements the CHOOSEINDEX algorithm: It uses the given random number generator to produce one floating point number between 0 and 1 and then searches for the interval that contains this number.

AffineIFS **1.7**
chooseIndex() **1.9**
extendedChaosGame() **1.9**

Algorithm 1.3

Generate a random integer with a given distribution. The return value is an integer between 1 and K, and the number k is returned with probability p_k.

CHOOSEINDEX$(p_1, \ldots, p_K)$ =

```
1   let u = random number between 0 and 1
2   var s = 0
3   for k = 1 to K
4       s = s + p_k
5       if u < s
6           return k
7   return K
```

Listing 1.9 ch1/AffineIFS

```
1  public List<Point> extendedChaosGame(
2          Point start, int iterations, Random rng) {
3      Point p = start;
4      var points = new ArrayList<Point>(iterations);
5      for (int i = 0; i < iterations; i++) {
6          points.add(p);
7          p = transforms[chooseIndex(rng)].transformPoint(p);
8      }
9      return points;
10 }
11
12 public int chooseIndex(Random rng) {
13     double u = rng.nextDouble();
14     double sum = 0.0;
15     for (int i = 0; i < probabilities.length - 1; i++) {
16         sum += probabilities[i];
17         if (u < sum)
18             return i;
19     }
20     return probabilities.length - 1;
21 }
```

Affine **1.5**
  transformPoint() **1.6**
AffineIFS **1.7**
  chooseIndex() 1.9
  extendedChaosGame() 1.9
ArrayList →
List →
Point **1.1**

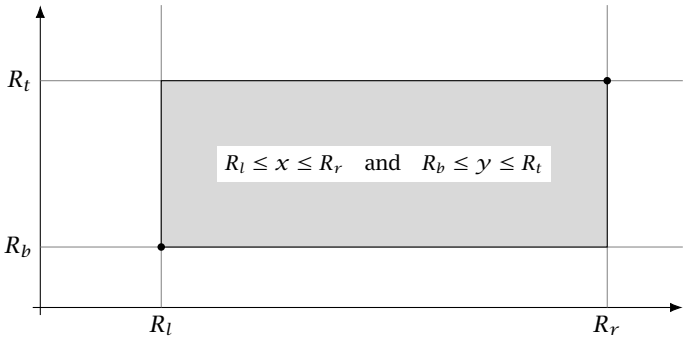


Figure 1.6 An axis-aligned rectangle.

1.3 Problems

Exercise 1.4. A simple geometric shape that is useful in many applications is the ***axis-aligned rectangle***, a rectangle whose sides are parallel to the coordinate axes. As shown in Fig. 1.6, each such a rectangle R is uniquely defined by the position of its four edges: the left and right edges R_l and R_r and the top and bottom edges R_t and R_b. The interior of such a rectangle consists of all points (x, y) that satisfy

$$R_l \le x \le R_r \qquad \text{and} \qquad R_b \le y \le R_t.$$

The rectangle is empty if there are no points that satisfy these relations; this is the case if either $R_l > R_r$ or $R_b > R_t$.

a) Given a set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$, how do you compute the smallest axis-aligned rectangle that contains all points?. (This rectangle is also referred to as the point set's ***bounding box***).

b) Given two axis-aligned rectangles A and B, how do you compute the smallest axis-aligned rectangle that encloses them both?

c) If two axis-aligned rectangles overlap, their intersection is also an axis-aligned rectangle. How do you compute its coordinates? The left (top) edge of the intersection must be the maximum of the two left (top) edges of the original rectangles, and the right (bottom) edge is the minimum of the two right (bottom) edges. These expressions also handles the special case in which the two rectangles don't overlap; in this case, the returned rectangle has a negative width or height and is therefore considered empty.

d) Listing 1.10 shows one possible way of defining a data type `Rectangle` for working with axis-aligned rectangles. Implement the missing methods.

`Rectangle` **C.2**

```
public record Rectangle(Point bottomLeft, Point topRight) {
    // The width and height of the rectangle.
    public int width();
    public int height();

    // Is the rectangle empty?
    public boolean isEmpty();

    // Determine whether 'p' lies inside the rectangle.
    public boolean contains(Point p);

    // Return intersection of 'this' and 'r'.
    public Rectangle intersection(Rectangle r);

    // The smallest rectangle that includes all points in the list.
    public static Rectangle boundingBox(List<Point> points);
}
```

Listing 1.10 Outline of a data type for representing axis-aligned rectangles.

Variations of the Chaos Game

Exercise 1.5. A simple heuristic for computing the probabilities associated with an affine IFS is based on ***determinants***. For a 2×2 matrix $\boldsymbol{A}$, the determinant $\det \boldsymbol{A}$ can be computed as follows:

$$\det \boldsymbol{A} = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ac - bd.$$

A fundamental result from linear algebra is that a linear transformation of the form

$$\boldsymbol{p} \mapsto \boldsymbol{A}\boldsymbol{p} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \boldsymbol{p}$$

scales the area of any shape by the factor $\det \boldsymbol{A}$. In the case of the Barnsley fern, for example, the areas of the four parallelograms shown in Fig. 1.4 are proportional to the determinants of the matrix parts of the corresponding functions. (If the determinant is negative, the transformation mirrors one of the coordinate axes; in this case, the *absolute value* of the determinant must be used.)

a) Compute the determinants of the four matrices that are used in the definition of the Barnsley fern. What do you observe?

b) Suggest a way to turn the values you obtain into probabilities that measure the footprint of each function.

Exercise 1.6. In the simplest version of the chaos game, which we discussed at the beginning of this chapter, the current point is always moved halfway toward a

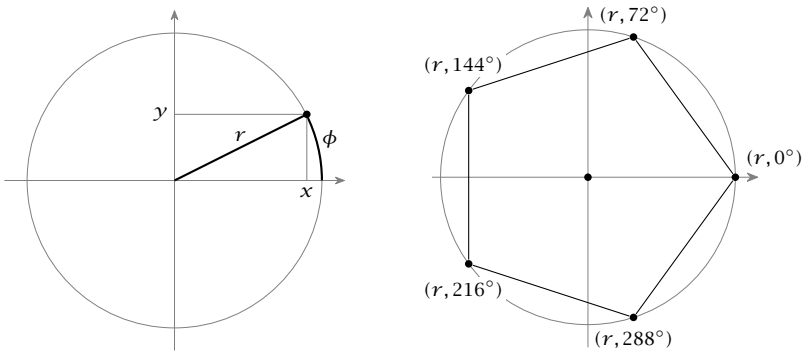
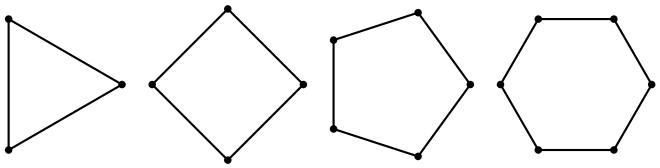


Figure 1.7 (Left) Points in the plane can be specified either using Cartesian coordinates x and y or polar coordinates, which indicate the distance from the origin r and the angle ϕ relative to the x axis. (Right) Polar coordinates make it easy to specify the corners of regular polygons.

random corner of a triangle. In this exercise we discuss a few interesting variations of this simple procedure.

The first variation is to replace the triangle with another regular polygon:



The easiest way to construct these polygons is to use ***polar coordinates***, which specify the position of a point using its distance r from the origin and the angle ϕ it makes with the x-axis, as illustrated in the left part of Fig. 1.7. Polar coordinates (r, ϕ) are related to the conventional Cartesian coordinates (x, y) by elementary trigonometry:

$$x = r\cos\phi, \qquad y = r\sin\phi. \tag{1.4}$$

a) Extend the `Point` type with a factory method called `polar()` that constructs a `Point` from its polar coordinates.

b) Explain how to perform the opposite conversion, from Cartesian to polar coordinates and implement two methods `Point.radius()` and `Point.angle()` that return a `Point`'s polar coordinates. *Hint:* to find the formula for ϕ, consider the ratio y/x.

The right part of Fig. 1.7 illustrates how polar coordinates can be used to compute the corners of a regular n-gon that is centered on the origin. The n corners of the polygon all have the same radius, whereas the angles are $360°/n$ degrees apart.

`Point` **1.1**
`angle()` **C.4**
`radius()` **C.4**

c) Implement a version of the chaos game that uses a regular n-gon as its base shape. Again, every step of the game should move the current point halfway towards a randomly chosen corners.

Another simple variation is to use a different rule for moving the current point. Instead of moving it *halfway* toward a corner, we can move it by a by a certain fraction α:

$$\boldsymbol{p} \quad \mapsto \quad \operatorname{lerp}(\boldsymbol{p}, \boldsymbol{c}_i, \alpha).$$

d) Experiment with different n-gons and different values of α and draw the resulting point sets. For which values of α does this version of the chaos game produce fractals?

Exercise 1.7. In Listing 1.4 we showed a simplified program for drawing the point sets produced by the chaos game. To complete this program, we need to create a suitable instance of `Graphics2D`, which is usually either a window on the screen or a bitmap image of type `BufferedImage`, and we need to scale and move the coordinates of the points so that they fit inside the window or image we are drawing into.

a) Write a program takes point sets generated by the chaos game and draws them, properly scaled, either to the screen or to a bitmap image.

b) Table 1.1 shows the parameters of four affine iterated function systems. Use the chaos game and the program developed in part (a) to generate images of the corresponding fractals.

Blending Iterated Function Systems

Exercise 1.8. Figure 1.8 shows a few frames from an animation that smoothly transforms the Barnsley fern on the left into the tree-shaped fractal on the right. The intermediate fractals are obtained by mixing the iterated function systems of the first and last fractal and are rendered using the standard chaos game. In this exercise we discuss how this is done.

Let's start with two arbitrary affine transformations of the form $f(\boldsymbol{p}) = \boldsymbol{F}\boldsymbol{p} + \boldsymbol{f}$ and $g(\boldsymbol{p}) = \boldsymbol{G}\boldsymbol{p} + \boldsymbol{g}$ and define a new function h that combines f and g using linear interpolation:

$$h(\boldsymbol{p}; t) = \operatorname{lerp}(f(\boldsymbol{p}), g(\boldsymbol{p}), t)$$

The parameter t is assumed to be a fixed number between 0 and 1. This new function is equal to f for $t = 0$, it is equal to g for $t = 1$, and it is a mixture of the two for every other value of t.

a) Show that h is also an affine transformation and can be written as

$$h(\boldsymbol{p}; t) = \operatorname{lerp}(\boldsymbol{F}, \boldsymbol{G}, t)\boldsymbol{p} + \operatorname{lerp}(\boldsymbol{f}, \boldsymbol{g}, t) \tag{1.5}$$

Here, the notation $\operatorname{lerp}(\boldsymbol{F}, \boldsymbol{G}, t)$ stands for the interpolation of the two matrices $\boldsymbol{F}$ and $\boldsymbol{G}$, which is obtained by interpolating its coefficients.

BufferedImage →

Table 1.1
Parameters of a few affine iterated function systems!examples.

Name	a	b	c	d	t_x	t_y	Probability
Tree	0.01	0	0	0.45	0	0	1/4
	0.42	-0.42	0.42	0.42	0	0.4	1/4
	0.42	0.42	-0.42	0.42	0	0.4	1/4
	-0.01	0	0	-0.45	0	0.4	1/4
Dragon curve	1/2	1/2	-1/2	1/2	0	0	1/2
	-1/2	1/2	-1/2	-1/2	1	0	1/2
Letters	0	-0.2	5/8	0	1/8	0	1/9
	3/8	0	0	0.2	3/16	1/2	1/9
	3/8	0	0	0.2	3/16	0	1/9
	0	-0.2	3/8	0	5/16	1/8	1/9
	3/8	0	0	0.2	5/8	1/2	1/9
	3/8	0	0	0.2	5/8	1/4	1/9
	3/8	0	0	0.2	5/8	0	1/9
	0	-0.2	1/8	0	3/4	3/8	1/9
	0	-0.2	1/8	0	3/4	1/8	1/9
Crystal	.255	0	0	.255	.3726	.6714	0.2
	.255	0	0	.255	.1146	.2232	0.15
	.255	0	0	.255	.6306	.2232	0.15
	.370	-.642	.642	.370	.6356	-.0061	0.75

Figure 1.8 Blending IFS fractals. The Barnsley fern on the left can be smoothly transformed into the tree-shaped fractal on the right. The intermediate fractals are obtained by interpolating the underlying iterated function systems.

Not only can we blend individual affine transformations, we can also blend *systems* of affine transformations. Assume that $F = (f_1, \ldots, f_n)$ and $G = (g_1, \ldots, g_n)$ are two iterated function systems of the same size n. We can now define a blended function system $H_t = (h_1, \ldots, h_n)$ linearly interpolating each pair of functions:

$$h_k(\boldsymbol{p}; t) = \operatorname{lerp}(f_k(\boldsymbol{p}), g_k(\boldsymbol{p}), t) \tag{1.6}$$

As we saw in part (a), each h_k is an affine transformation, so the system H is another affine IFS that can be rendered using the chaos game of this chapter. The parameter t indicates the time that has passed since the start of the animation: We start with the original function system $F = H_0$ at $t = 0$ and end with the target function system $G = H_1$ at $t = 1$. For intermediate values of t, the fractals generated by H_t are a mixture of those generated by F and G.

b) Write a program that uses Eq. (1.6) to generate animations of iterated function systems similar to the one shown in Fig. 1.8. (Refer to Table 1.1 for the parameters of fractals you can use for testing.) How do you blend the tables of probabilities and how do you handle the case where the two function systems don't contain the same number of functions?

1.4 Chapter Notes

Iterated function systems are a popular means of constructing fractal shapes: They are easy to describe, easy to program, and can be used to model infinitely detailed geometric shapes using just a few parameters. See Paul Bourke's website (`https://paulbourke.net/fractals/ifs/`) for an overview of shapes that can be generated in this way.

It is possible to extend the simple IFSs discussed in this chapter to produce still images and animations that are even more striking. The best-known example is the so-called ***fractal flame algorithm***, which adds new function types, colorization, as well as additional transformation and post-processing steps to produce and endless variety of images and animations such as the one shown in Fig. 1.9 [31]. The *Electric Sheep* screen saver (`https://electricsheep.org/`) is based on this algorithm.

There is a deep and beautiful mathematical theory behind iterated function systems that concerns itself with questions such as:

- Under which conditions does the chaos game converge?
- How are the functions in the IFS related to the resulting fractal?
- Why does the chaos game always produce the same fractal even if we change the starting point or make different random choices during the process?

For details, see the books by Barnsley [10] and Peitgen et al. [76].

To give you a taste of this theory, let us briefly discuss one of its main results: The chaos game converges only if the functions are ***contractive***, which means

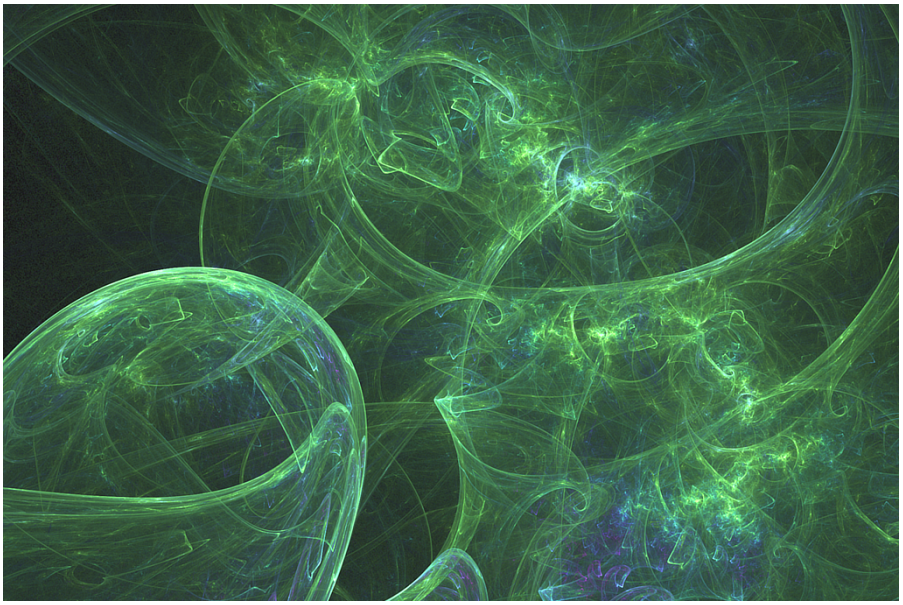


Figure 1.9 An image generated using the fractal flame algorithm.

that they make everything smaller. In mathematical terms, a function f is called contractive if the points $f(\boldsymbol{p})$ and $f(\boldsymbol{q})$ are closer together than $\boldsymbol{p}$ and $\boldsymbol{q}$ for every possible choice of $\boldsymbol{p}$ and $\boldsymbol{q}$:

$$|f(\boldsymbol{p}) - f(\boldsymbol{q})| < |\boldsymbol{p} - \boldsymbol{q}| \qquad \text{for all } \boldsymbol{p} \text{ and } \boldsymbol{q}.$$

The upper bound of the ratio $|f(\boldsymbol{p}) - f(\boldsymbol{q})|/|\boldsymbol{p} - \boldsymbol{q}|$ is called the ***contraction factor*** of the function f. A function is therefore contractive if its contraction factor is less than 1.

Consider the function $\boldsymbol{p} \mapsto (\boldsymbol{p} + \boldsymbol{c})/2$ that we used at the beginning of this chapter to move points halfway towards the corner $\boldsymbol{c}$. It's easy to show that this function is contractive with a contraction factor of $1/2$:

$$|f(\boldsymbol{p}) - f(\boldsymbol{q})| = |(\boldsymbol{p} + \boldsymbol{c})/2 - (\boldsymbol{q} - \boldsymbol{c})/2| = |\boldsymbol{p} - \boldsymbol{q}|/2.$$

Therefore, any set of functions that move their argument halfway toward a fixed set of points forms a convergent IFS.

It is also possible to derive the contraction factors of arbitrary affine transformations, but a few concepts from linear algebra are required. The main result is that the contraction factor of

$$f(\boldsymbol{p}) = \boldsymbol{A}\boldsymbol{p} + \boldsymbol{v},$$

is the square root of the largest eigenvalue of $\boldsymbol{A}^t\boldsymbol{A}$, where $\boldsymbol{A}^t$ is the transpose of the matrix $\boldsymbol{A}$. For instance, in the case of the Barnsley fern, the contraction factors of the functions $f_0, \ldots, f_3$ are 0.16, 0.85, 0.32, and 0.34.

Chapter 6

Exploring Peg Solitaire

No one knows how many different ways there are
to solve the puzzle leaving
the last counter in the center.
— Martin Gardner [40]

Peg solitaire is a well-known board game with a long history. Today, the most common version of the game is played on a board with 33 holes that are arranged in cross-like shape as shown in Fig. 6.1. At the start of the game, every hole except for the one in the center contains a marble or a wooden peg, and the goal is to find a sequence moves that inverts the board so that only a single peg in the center remains.

The only allowed move in peg solitaire is to jump one of the pegs over an adjacent peg into a hole. The peg that was jumped over is removed from the

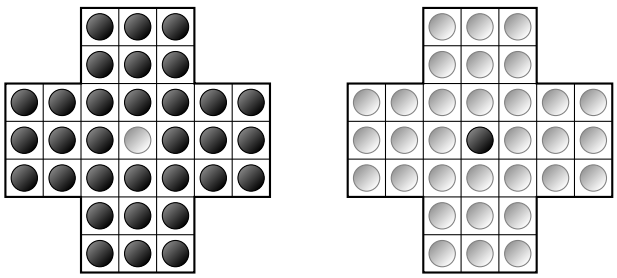


Figure 6.1 Peg solitaire. The goal is to remove all pegs from the board so that only the one in the center remains.

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

Figure 6.2 Numbering scheme for pegs. To simplify index computations, the 16 positions in the corners are also enumerated.

board and creates a new hole, so each move transforms a vertical or horizontal arrangement of the form "XXO" (two pegs and one hole) into "OOX" (two holes and one peg). Since each jump removes one peg, every valid solution requires exactly 31 jumps. The question is: Are there any solutions? If yes, how many? And how do we find them?

6.1 Representing Boards

Let's start with a simple problem: How do we store the state of the game? One solution is to enumerate the board as shown in Fig. 6.2 and represent the positions of pegs or holes as sets of integers. For example, the set of all valid positions on the board can be represented by the following set:

$$\text{(full board)} = \{2, 3, 4, 9, 10, 11, 14, \ldots, 34, 37, 38, 39, 44, 45, 46\}$$

In the starting configuration, there is a peg in every hole except the one in the center, so the corresponding set consists of all valid positions except the 24th:

$$\text{(initial board)} = \text{(full board)} - \{24\}. \tag{6.1}$$

Likewise, the final state of the game has a single peg in the center and therefore has the set representation $\{24\}$. At the start of the game, the only possible move is to jump to the empty hole in the center, and the only four pegs that can jump there are the ones at positions 10, 22, 26, and 38; the set of pegs that can be moved can be written as $\{10, 22, 26, 38\}$.

One advantage of this representation is that every such set can be stored very compactly a sequence of 49 bits in which the kth bit indicates whether there is a peg at position k. (Of course, set of *holes* can be represented in the same way.) The term ***bitboard*** is often used for the resulting data structure. We collect a few basic functions for working with these bitboards in a class called `PegBoard`, which is outlined in Listing 6.1. We start by defining three constants: `FULL_BOARD` is the set

`PegBoard` **6.1**

of all valid positions on the board, `INIT_PEGS` is the initial configuration of pegs, and `FINAL_PEGS` is the desired final configuration of pegs.

Listing 6.1 ch6 / PegBoard

```
public class PegBoard {
    public static long FULL_BOARD = Bits.range(2, 5) | Bits.range(9, 12)
            | Bits.range(14, 35) | Bits.range(37, 40) | Bits.range(44, 47);
    public static long INIT_PEGS = FULL_BOARD & ~Bits.bit(24);
    public static long FINAL_PEGS = Bits.bit(24);

    // ...
}
```

How can we perform a single jump in this representation? Assume we are given the current position of all pegs as a bit set `board` and two integers `peg` and `hole` that indicate which peg to move into which hole. With our numbering scheme, we can compute the index `mid` of the peg that is jumped over by averaging `peg` and `hole`. The resulting board is obtained by removing the two pegs at `peg` and `mid` and adding a new peg at `hole`; alternatively, we can flip the corresponding three bits using a single exclusive OR operation, as shown in Listing 6.2.

Listing 6.2 ch6 / PegBoard

```
public static long jump(long board, int peg, int hole) {
    int mid = (peg + hole) / 2;
    return board ^ Bits.bits(peg, mid, hole);
}
```

Bitboards

We now come to the main advantage of using bit sets to store sets of board positions, namely the ability to use bit operations to modify all set elements at once. Consider the problem of moving individual pegs or groups of pegs. We can move a single peg to the right or left by adding ± 1 to its index and up or down by adding ± 7. If the positions of multiple pegs are stored in a bitboard, we can move them all at once by shifting the bit pattern by ± 1 or ± 7 places!

This trick works for most but not all positions on the board. For example, if you try to move the peg at position 21 to the left by subtracting 1, you end up with position 20 on the opposite side of the board. In general, we have to be careful with positions on the boundary of the board.

Fortunately, bitboards give us an easy way to handle this special case as well: All we have to do is remove the problematic positions before performing the bit shift. For example, we can safely move a bitboard to the left if we first remove the positions on the left boundary, which can be done by subtracting the set $\{2, 9, 14, 21, 28, 37, 44\}$. The `left()` function in Listing 6.3 implements this operation, using the expression `board & ~BOUNDARY_LEFT` to compute the set difference

Bits
bit() 5.5
bits() 5.5
range() 5.6
PegBoard 6.1
FINAL_PEGS 6.1
FULL_BOARD 6.1
INIT_PEGS 6.1
jump() 6.2
left() 6.3

of `board` and `BOUNDARY_LEFT`. If `p` is a set of positions on the board, `left(p)` is the set of positions to the left of `p` and `left(left(p))` the set of positions two steps left of `p`.

Listing 6.3 ch6 / PegBoard

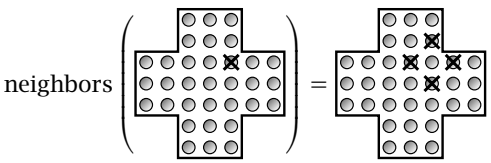
```
1  static long BOUNDARY_LEFT = Bits.bits(2, 9, 14, 21, 28, 37, 44);
2  static long left(long board) {
3      return (board & ~BOUNDARY_LEFT) >>> 1;
4  }
5
6  static long BOUNDARY_RIGHT = Bits.bits(4, 11, 20, 27, 34, 39, 46);
7  static long right(long board) {
8      return (board & ~BOUNDARY_RIGHT) << 1;
9  }
10
11 static long up(long board) {
12     return (board >>> 7) & FULL_BOARD;
13 }
14
15 static long down(long board) {
16     return (board << 7) & FULL_BOARD;
17 }
```

In a similar manner, we can define functions `right()`, `up()` and `down()` that move game boards in the other three directions. For the implementation of `right()`, we replace the right shift with a left shift and remove bits on the opposite boundary. Moving the board up and down can be accomplished by shifting the bitboard 7 places to the left or right. In this case, it isn't possible for bits to wrap around to the opposite side of the board, so can simply remove invalid positions from the bitboard after shifting the bits.

We can use these four functions to define more advanced operations on bitboards. For instance, given a set of `positions`, the set of all neighboring positions on the board is the union of positions obtained by moving in the four possible directions:

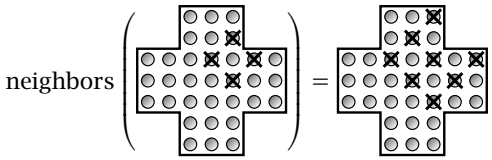
```
static long neighbors(long positions) {
    return left(positions) | right(positions)
        | up(positions) | down(positions);
}
```

If `positions` contains just a single bit, `neighbors()` computes its direct neighbors:



Bits
  bits() 5.5
PegBoard 6.1
  BOUNDARY_LEFT 6.3
  BOUNDARY_RIGHT 6.3
  down() 6.3
  left() 6.3
  right() 6.3
  up() 6.3

But if multiple bits are set, it returns the union of *all* direct neighbors:



That's the main benefit of bitboards: They make it possible to express many computations involving sets of positions using just a handful of bit operations.

Computing Possible Moves

We can also use bitboards to quickly compute the set of pegs that are currently movable, that is, the set of pegs that are two places away from a hole with another peg between them. Let's start with a single direction first: Given a set `pegs` that contains all pegs that are currently on the board, which subset of these pegs can jump to the left?

First we compute the subset of pegs that have another peg to their left. This is obviously the same as the subset of pegs that are to the right of another peg, which we can compute by moving all pegs to the right and computing the intersection with the original pegs:

```
pegs & right(pegs)
```

It's helpful to mentally translate expressions of the form `pegs & X` into "pegs that have some property `X`." Here, the mental translation is "pegs that are to the right of another peg."

We can use the same idea to compute the set of pegs that are to the right of a hole. The set of holes can be computed by subtracting `pegs` from `FULL_BOARD`, which effectively inverts the current board:

```
long holes = FULL_BOARD & ~pegs;
```

We then compute the set of pegs directly to the right of a hole by moving all holes and computing the intersection with `pegs`:

```
pegs & right(holes)
```

By combining the two expressions we obtain the set of pegs that are to the right of a peg that is to the right of a hole — in other words, the set of pegs that can jump to the left.

```
pegs & right(pegs & right(holes))
```

In the same way, we can compute the sets of pegs that can jump in any of the three other directions, and the union of these four sets gives us the set of all pegs that can be moved:

PegBoard **6.1**
FULL_BOARD **6.1**
right() **6.3**

```
(pegs & down(pegs & down(holes)))
    | (pegs & up(pegs & up(holes)))
    | (pegs & right(pegs & right(holes)))
    | (pegs & left(pegs & left(holes)))
```

This expression can be simplified further by "factoring out" the common term "`pegs &`". This is possible because the bit operators '`|`' and '`&`' satisfy ***distributive laws*** similar to those of addition and multiplication; see Exercise 6.2. This gives us the final expression for the subset of movable pegs:

```
movablePegs = pegs & (down(pegs & down(holes))
    | up(pegs & up(holes))
    | right(pegs & right(holes))
    | left(pegs & left(holes)))
```

Another problem can be solved in almost the same way: Given a single peg on the board, how can we determine where it can jump to? To test whether the peg can jump in a particular direction, we simply check whether there is first another peg and then a hole in that direction. Repeat this for all four directions and you have the list of possible moves. Alternatively, we can again use bit operations to determine all possible destinations in one go. If `position` is a bit set that contains just the peg we are interested in, we can compute the set of reachable holes using an expression that is almost identical to the one we used to compute the set of movable pegs:

```
reachableHoles = holes & (down(pegs & down(position))
    | up(pegs & up(position))
    | right(pegs & right(position))
    | left(pegs & left(position)))
```

This expression moves `position` in all four directions, keeping only those positions that have another peg at distance 1 and a hole at distance 2. The result is a bit set containing all holes that are reachable from `position`. (If `position` doesn't contain a single peg but an arbitrary subset of `pegs`, the expression returns the union of all reachable holes.)

As you can see, the bit expressions for `movablePegs` and `reachableHoles` are almost identical, so we factor out the common part that computes the reachable positions that are one jump away, which gives us the function `reachablePositions()` shown in Listing 6.4. Using this function, we can then easily compute the set of pegs that are currently movable and the holes that are reachable from a set of positions. The set of movable pegs is simply the subset of pegs that are one jump away from a hole (`movablePegs()`), and the set of reachable holes is the subset of holes that are one jump away from the positions of interest (`reachableHoles()`).

Using these two functions, we can now analyze a given game state and determine all possible moves. For example, consider the game state shown in Fig. 6.3. To compute the possible moves for this state, we first call `movablePegs()` to obtain the positions of all pegs for which *any* move exists, and then use `reachableHoles()`

PegBoard **6.1**
down() **6.3**
left() **6.3**
movablePegs() **6.4**
reachableHoles() **6.4**
reachablePositions() **6.4**
right() **6.3**
up() **6.3**

Listing 6.4 ch6 / PegBoard

```
// Compute all positions that can be reached from the positions
// in 'start' by jumping over an adjacent peg.
static long reachablePositions(long pegs, long start) {
    return down(pegs & down(start))
            | up(pegs & up(start))
            | right(pegs & right(start))
            | left(pegs & left(start));
}

// Compute the subset of 'pegs' that are movable.
public static long movablePegs(long pegs) {
    long holes = FULL_BOARD & ~pegs;
    return pegs & reachablePositions(pegs, holes);
}

// Compute the set of holes that can be reached from the pegs in 'start'.
public static long reachableHoles(long pegs, long start) {
    long holes = FULL_BOARD & ~pegs;
    return holes & reachablePositions(pegs, start);
}
```

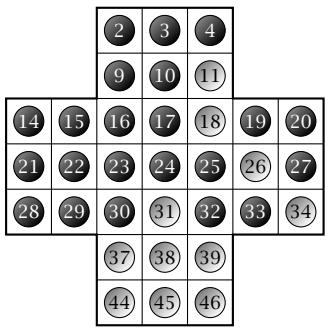


Figure 6.3 An intermediate game state in peg solitaire.

PegBoard **6.1**
FULL_BOARD **6.1**
down() **6.3**
left() **6.3**
movablePegs() 6.4
reachableHoles() 6.4
reachablePositions() 6.4
right() **6.3**
up() **6.3**

to determine the eligible target positions for each of them:

```
long pegs = Bits.bits(
        2, 3, 4, 9, 10, 14, 15, 16, 17, 19, 20, 21, 22, 23,
        24, 25, 27, 28, 29, 30, 32, 33);
for (int movable : Bits.iterate(PegBoard.movablePegs(pegs))) {
    long reachable = PegBoard.reachableHoles(pegs, Bits.bit(movable));
    System.out.printf("%d: %s%n", movable,
            Bits.toList(reachable));
}
```

This outputs the following list of positions:

```
9: [11]
16: [18]
17: [31]
20: [18, 34]
23: [37]
24: [26]
25: [39]
29: [31]
32: [18, 34]
33: [31]
```

The first number in each row is the position of a movable peg, and the list of numbers in square brackets are the holes this peg can reach in a single jump. For example, the pegs at positions 20 and 32 can both jump to the holes at positions 18 and 34, whereas only a single move is available for the pegs at positions 9 and 16.

Exercises

Exercise 6.1. What does the following expression compute, when `board` is a bit set of peg positions?

```
reachableHoles(board, movablePegs(board))
```

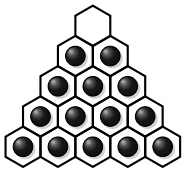
⋆ **Exercise 6.2.** Prove the following distributive laws that hold for arbitrary bit patterns A, B, and C:

$$A \,\&\, (B \mid C) = (A \,\&\, B) \mid (A \,\&\, C)$$
$$A \mid (B \,\&\, C) = (A \mid B) \,\&\, (A \mid C).$$

Hint: Consider individual bits first.

Exercise 6.3. The following image shows a variation of peg solitaire that is played on a triangular board:

Bits
bit() **5.5**
bits() **5.5**
iterate() **5.9**
toList() **5.9**
PegBoard **6.1**
movablePegs() **6.4**
reachableHoles() **6.4**
System →



As before, pegs are removed by jumping over adjacent pegs, but now there are six directions of movement.

a) Find a solution to the board shown above.

b) Explain how to store this board using bits and how to compute all legal jumps.

Exercise 6.4. Connect 4 is a two-player game that is played on an upright board that consists of 7 columns and 6 rows. Both players start with 21 tokens of a certain color, say white for the first player and black for the second, and take turns dropping tokens into the columns of the board. The first player to complete a vertical, horizontal, or diagonal line of four tokens wins the game. Figure 6.4 shows a game state in which the white player has won after eight moves.

Explain how to represent the game state using bitboards and how to quickly determine whether there are four identically colored tokens in a line somewhere on the board.

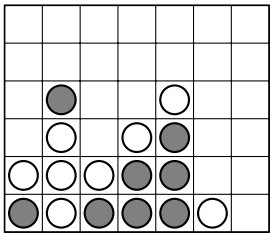


Figure 6.4 A game state of *Connect 4*. The white player has just won by completing a diagonal line of four tokens.

6.2 Backtracking

Now that we know how to represent boards and analyze possible moves, we can turn to the problem of assembling these moves into solutions that lead from `INIT_BOARD` to `FINAL_BOARD`. The simplest algorithm for systematically finding such solutions is based on a recursive strategy known as ***backtracking***.

Backtracking is a general procedure for generating all possible arrangements of certain objects — in our case, the various sequences of moves that can be performed in peg solitaire. Every backtracking algorithm has the same overall structure: Given a partial solution `current`, the idea is to use recursion to exhaustively explore all possible solutions that start with `current`. In pseudocode, this looks as follows:

```
backtrack(current) {
    if (current == desired solution) {
        print("Solution found: " + current)
    } else {
        for (next : states reachable from current)
            backtrack(next)
    }
}
```

The use of recursion ensures that returning from the function automatically "backtracks" to the last partial solution that needs to be explored further.

It's easy to adapt this general scheme to our problem of generating all possible solutions of peg solitaire. Each partial solution `current` is now the current state of the board, and the states that are reachable from `current` are the boards that are obtained by making a single jump:

```
solveBoard(current) {
    if (current == FINAL_BOARD) {
        print("Solution found")
    } else {
        for (next : boards reachable from current in a single jump)
            solveBoard(next)
    }
}
```

As illustrated in Fig. 6.5, this backtracking algorithm explores the possible states of the game in a hierarchical manner and forms a tree as it progresses; since the full tree is astronomically large, we only show a few of its nodes. Each node corresponds to a particular game state, the children of each node are the states that can be reached by making a single jump, and edges between nodes indicate recursive function calls made by the algorithm. Backtracking starts at the root of the tree on the left and picks one of the four reachable states, such as the one labeled *A*. It then explores *A* and the entire subtree beneath it recursively. Once that is done, the algorithm returns to the game states *B*, *C*, and *D* in the second column, which are then explored in the same manner.

> When backtracking is used to traverse tree-like structures, the resulting algorithm is also referred to as ***depth-first search*** since each subtree is explored "in depth" before any of its siblings. In the following chapter, we will examine a related technique called *breadth-first search*, which can be used to explore trees level by level.

Listing 6.5 shows one possible implementation of the backtracking algorithm for solving peg solitaire. We use two nested `for` loops to generate the possible jumps: The first one iterates over the movable pegs and the second one over the reachable holes for each peg. We then try each possible jump recursively. The

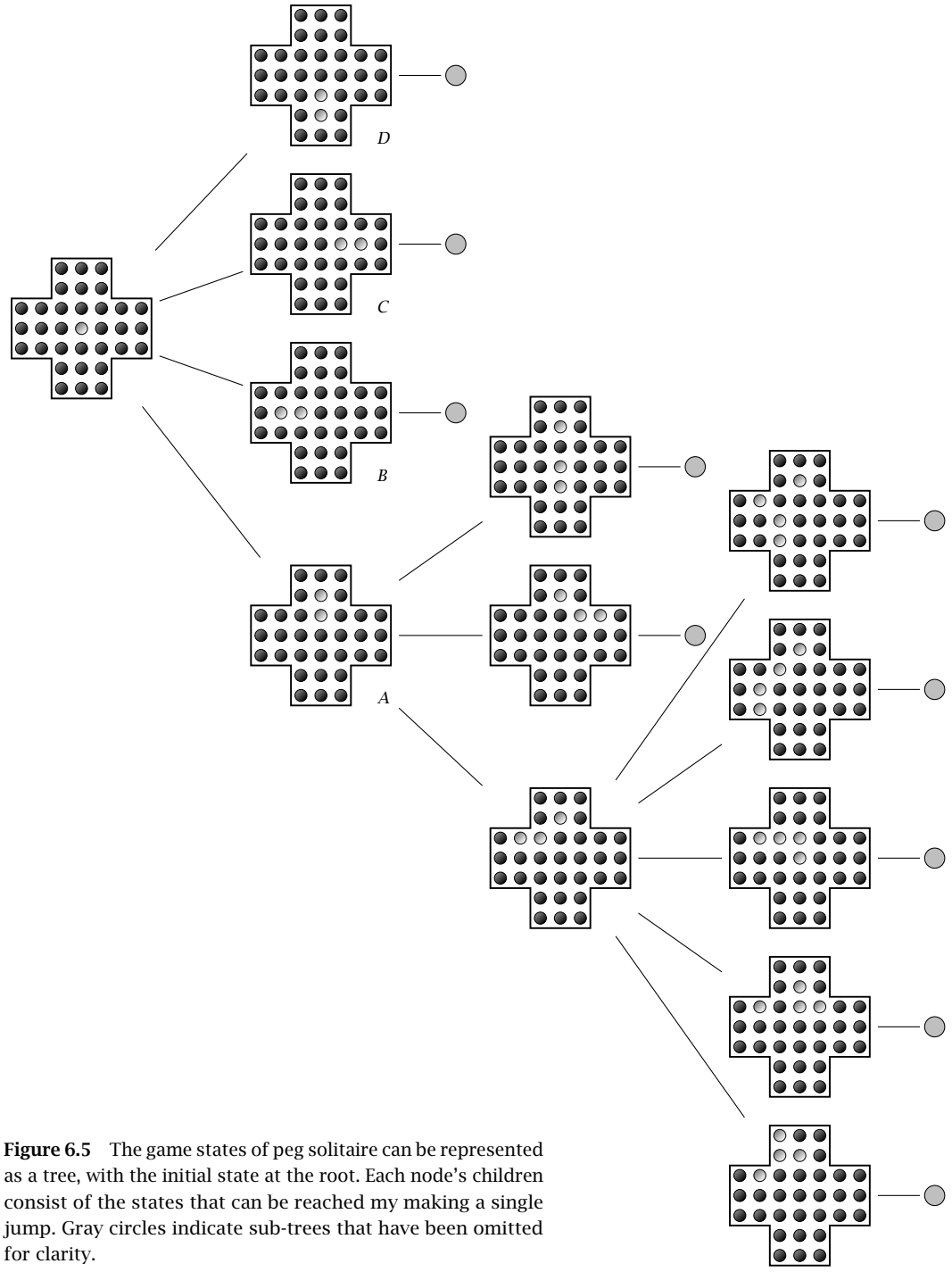


Figure 6.5 The game states of peg solitaire can be represented as a tree, with the initial state at the root. Each node's children consist of the states that can be reached my making a single jump. Gray circles indicate sub-trees that have been omitted for clarity.

member variable `jumpList` stores the current sequence of jumps as a nested list of integers; each entry in this list is a pair of integers (`peg`, `hole`) that indicates which peg was moved to which hole. New entries are added to `jumpList` before each recursive call to `printSolutions()` and removed immediately afterward.

Listing 6.5 ch6 / PegSolve

```
private final List<List<Integer>> jumpList = new ArrayList<>();

public void printSolutions(long pegs) {
    if (pegs == PegBoard.FINAL_PEGS) {
        System.out.println(jumpList);
        return;
    }
    for (int movable : Bits.iterate(PegBoard.movablePegs(pegs))) {
        long reachable = PegBoard.reachableHoles(pegs, Bits.bit(movable));
        for (int hole : Bits.iterate(reachable)) {
            jumpList.add(List.of(movable, hole));
            printSolutions(PegBoard.jump(pegs, movable, hole));
            jumpList.remove(jumpList.size() - 1);
        }
    }
}
```

We can now solve peg solitaire by calling `printSolutions()` with `INIT_PEGS` as the starting configuration:

```
new PegSolve().printSolutions(PegBoard.INIT_PEGS);
```

The program quickly finds the first solution and continues to generate several hundred solutions per second. Here are the first two lines of the output:

```
[[10, 24], [15, 17], [2, 16], [4, 2], [17, 15], [14, 16], [18, 4],
 [20, 18], [23, 9], [2, 16], [21, 23], [23, 9], [25, 11], [4, 18],
 [27, 25], [25, 11], [37, 23], [28, 30], [30, 16], [9, 23],
 [23, 25], [32, 18], [11, 25], [34, 32], [31, 33], [46, 32],
 [25, 39], [44, 46], [46, 32], [33, 31], [38, 24]]
[[10, 24], [15, 17], [2, 16], [4, 2], [17, 15], [14, 16], [18, 4],
 [20, 18], [23, 9], [2, 16], [21, 23], [23, 9], [25, 11], [4, 18],
 [27, 25], [25, 11], [37, 23], [28, 30], [30, 16], [9, 23],
 [23, 25], [32, 18], [11, 25], [34, 32], [31, 33], [46, 32],
 [44, 46], [25, 39], [46, 32], [33, 31], [38, 24]]
...
```

As you can see, the two solutions are almost identical and differ only in the order of the last few jumps. This is a general characteristic of peg solitaire: Once you have found one solution, you can usually generate many more by reordering jumps that are independent of each other.

Bits
  bit() **5.5**
  iterate() **5.9**
List →
PegBoard **6.1**
  FINAL_PEGS **6.1**
  INIT_PEGS **6.1**
  jump() **6.2**
  movablePegs() **6.4**
  reachableHoles() **6.4**
PegSolve
  printSolutions() 6.5
System →

6.3 Counting Solutions

As we saw in the previous section, solving peg solitaire using backtracking is fairly easy. In the following we will see why this is the case: There are so many solutions that it's hard *not* to stumble upon them while mindlessly trying all possible moves. How many solutions are there exactly? Let's find out.

We start with a simple variation of the backtracking algorithm used in the previous section, except that we now count the solutions instead of printing them:

```
int countSolutions(current) {
    if (current == FINAL_BOARD) {
        return 1
    } else {
        int n = 0
        for (next : boards reachable from 'current' in a single jump)
            n += countSolutions(next)
        return n
    }
}
```

The function returns the number of solutions that are reachable from `current`. If `current` is itself the solution, `countSolutions()` returns 1. Otherwise, it counts the number of solutions that start with one of the possible next jumps and then returns their sum.

Unfortunately, peg solitaire has so many solutions that this simple version of `countSolutions()` doesn't complete in any reasonable amount of time. To make the problem more tractable, we will use two techniques: ***normalization***, which simplifies computations by choosing a single, unique representation for game states that are equivalent; and ***memoization***, which is a general method for speeding up recursive computations by tabulating the results of previous computations. Let's see how they work and why they are so effective in this case.

Normalization

Many puzzles have symmetries we can exploit to reduce the number of moves or alternatives we have to consider. This is also true for peg solitaire. For example, the four different boards that can be reached with the first jump (shown in the second level of the tree in Fig. 6.5) differ only in their orientation, so each of them must lead to the same number of solutions. We can therefore speed up `countSolutions()` by choosing one of the possible four jumps in the first step, computing the resulting number of possible solutions, and then returning four times this number — without investigating the other three possible jumps at all.

It's easy to generalize this idea to any board configuration. We first note that rotation isn't the only kind of symmetry we can exploit: All boards that can be obtained by rotation, vertical or horizontal reflection, or any combination thereof have the same number of solutions, provided we are searching for a solution with the final peg in the center of the board. As illustrated in Fig. 6.6, this gives us

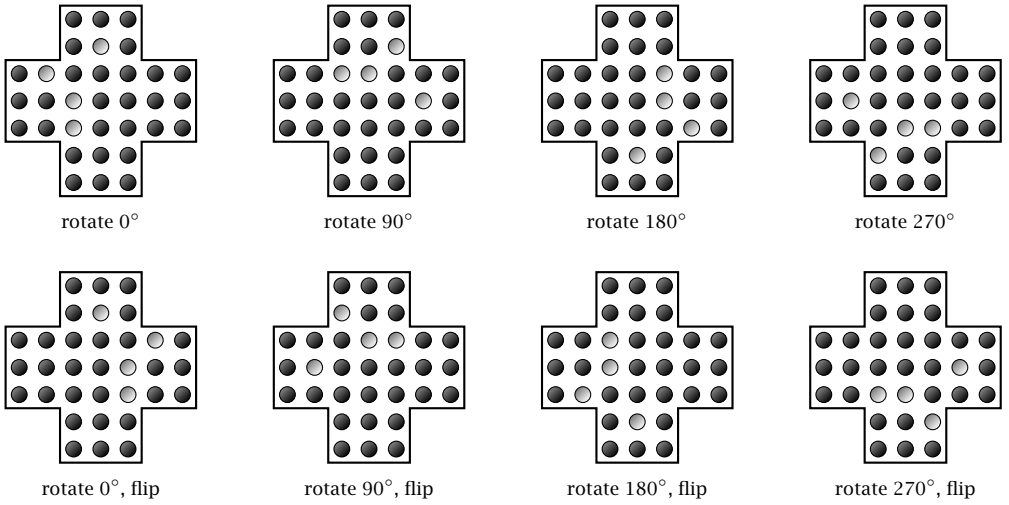


Figure 6.6 .
]Symmetries in peg solitaire. In a standard peg solitaire game, all eight boards shown here are equivalent in the sense that they lead to solutions that differ only in their orientation.

a maximum of eight configurations that can be considered equivalent: the four rotations we already discussed and their reflections. (Both horizontal and vertical reflections produce the same four states in the bottom row, albeit in a different order.)

The idea behind normalization is to choose a unique instance from the eight configurations in Fig. 6.6, thereby making it easier to detect boards that have already been investigated. For boards that are stored as bit sets, the easiest way to choose a unique representative is to compute the minimum of the corresponding integers. The `normalize()` method in Listing 6.6 performs this normalization by rotating and mirroring a given bitboard in all possible ways and keeping track of the one that is numerically smallest. In our case, the numerically smallest bitboard is the one with holes at the largest indices; in Fig. 6.6 this is the board in the lower right.

To implement `rotate()` and `flipH()`, we first note that both operations can be described by moving each bit to a new position. We use two arrays to specify the new positions of each bit (Listing 6.7): `FLIP_H` mirrors each row of the board horizontally and `ROTATE` rotates it counterclockwise. The kth entry in each array indicates the target position of the kth peg. After defining these permutation tables, we reorder the bits by calling `shuffleBits()`, which is defined in Listing 6.8.

What do we do with the normalized boards we just constructed? This is where the second technique, *memoization*, comes in.

```
PegCount
  FLIP_H 6.7
  ROTATE 6.7
  flipH() 6.7
  normalize() 6.6
  rotate() 6.7
  shuffleBits() 6.8
```

Listing 6.6 ch6 / PegCount

```
// Map 'board' to a unique representative from all equivalent
// rotated and flipped boards.
static long normalize(long board) {
    long result = Math.min(board, flipH(board));
    long rot90 = rotate(board);
    result = Math.min(result, Math.min(rot90, flipH(rot90)));
    long rot180 = rotate(rot90);
    result = Math.min(result, Math.min(rot180, flipH(rot180)));
    long rot270 = rotate(rot180);
    result = Math.min(result, Math.min(rot270, flipH(rot270)));
    return result;
}
```

Listing 6.7 ch6 / PegCount

```
private static byte[] FLIP_H = {
        6, 5, 4, 3, 2, 1, 0,
        13, 12, 11, 10, 9, 8, 7,
        20, 19, 18, 17, 16, 15, 14,
        27, 26, 25, 24, 23, 22, 21,
        34, 33, 32, 31, 30, 29, 28,
        41, 40, 39, 38, 37, 36, 35,
        48, 47, 46, 45, 44, 43, 42
};

private static byte[] ROTATE = {
        6, 13, 20, 27, 34, 41, 48,
        5, 12, 19, 26, 33, 40, 47,
        4, 11, 18, 25, 32, 39, 46,
        3, 10, 17, 24, 31, 38, 45,
        2, 9, 16, 23, 30, 37, 44,
        1, 8, 15, 22, 29, 36, 43,
        0, 7, 14, 21, 28, 35, 42
};

public static long flipH(long board) {
    return shuffleBits(board, FLIP_H);
}

public static long rotate(long board) {
    return shuffleBits(board, ROTATE);
}
```

Math →
  min()
PegCount
  FLIP_H 6.7
  ROTATE 6.7
  flipH() 6.7
  normalize() 6.6
  rotate() 6.7
  shuffleBits() **6.8**

Listing 6.8 ch6 / PegCount

```
1  // Reorder bits by moving each one to a new position.
2  private static long shuffleBits(long bits, byte[] targetIndex) {
3      long result = 0L;
4      for (int i : Bits.iterate(bits))
5          result |= Bits.bit(targetIndex[i]);
6      return result;
7  }
```

Memoization

Memoization is a programming technique for speeding up expensive functions by caching the results from previous calls in an auxiliary data structure. Effectively, memoization is a method for tabulating a given function *on demand*: we don't have to know beforehand which function arguments actually occur in practice but simply tabulate those that *do* occur *when* they occur.

Memoization can be used for both recursive and non-recursive functions. If `F` is a non-recursive function that depends on a single parameter `x`, we can create a memoized version of this function by wrapping it in a new function `memoizeF` that uses an auxiliary data structure `cache` to quickly return the results of previous calls without repeating the actual computation:

```
memoizeF(x) {
    if (!cache.contains(x))
        cache[x] = F(x)
    return cache[x]
}
```

If `cache` already contains a value for the argument `x` it is returned directly; otherwise, the original function `F(x)` is called and `cache` is updated with the result.

For recursive functions, we have to interleave the memoization logic with the implementation of the recursive function. It's easiest to see how this works by considering a concrete example. The factorial function $n!$ is defined recursively as

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

To implement a memoized version of this function, we first define a hash map `cache` to store previous results and then define a `factorial()` method as shown in the following listing:

```
Bits
  bit() 5.5
  iterate() 5.9
PegCount
  shuffleBits() 6.8
```

```
static HashMap<Integer, Integer> cache = new HashMap<>();
static int factorial(int n) {
    if (n == 0)  // base case
        return 1;
    if (!cache.containsKey(n))
        cache.put(n, n * factorial(n - 1));
    return cache.get(n);
}
```

As you can see, the memoized version of a recursive function is still recursive, but redundant function calls are avoided by caching previous results.

In principle, memoization be used with any ***pure function***, that is, a function whose return value depends only on its arguments and that doesn't have any *side effects*. In addition, memoization works best if the function is sufficiently expensive to compute and if it is called repeatedly with the same arguments: There is little value in tabulating simple arithmetic expressions or functions that are called only once. The main downside of the technique is that that the tables produced by memoization can consume a significant amount of memory.

Let's see how to use memoization to speed up the `countSolutions()` function for counting the solutions of peg solitaire. Both prerequisites mentioned above are satisfied: The function is expensive to compute (at least for boards that contain more than a handful of pegs) and the same boards are evaluated over and over again, both naturally (because there are often several different sequences of moves that lead from one board to another), and even more so if we also use normalization to merge game states that are considered equivalent. To implement memoization, we use a hash table that maps boards (long integers in bit representation) to the number of possible solutions (because this number can grow large, we use long integers here as well). The resulting implementation of `countSolutions()` is shown in Listing 6.9.

We can now determine the total number of solutions by calling

```
countSolutions(PegBoard.INIT_PEGS);
```

On the author's computer, the function completes in just a few minutes. The result? For the standard peg solitaire board there are 40 861 647 040 079 968, or approximately 41 quadrillion, unique solutions.

This raises an interesting questions: If solutions are so plentiful, why do human players still find peg solitaire moderately difficult? The main reason is (ironically) that the number of solutions is *only* 41 quadrillion — a tiny number compared to the vast number of possible ways to play the game. As we will discuss in Exercise 6.6 on the next page, most boards that are reachable from the starting configuration are unwinnable. If you play peg solitaire without some kind of strategy, you are therefore likely to end up in a dead end.

PegBoard **6.1**
INIT_PEGS **6.1**
PegCount
countSolutions() **6.9**

Listing 6.9 ch6 / PegCount

```
private static HashMap<Long, Long> numSolutions = new HashMap<>();

public static long countSolutions(long board) {
    if (board == FINAL_PEGS)
        return 1;
    long normalized = normalize(board);
    if (numSolutions.containsKey(normalized)) {
        return numSolutions.get(normalized);
    } else {
        long n = 0;
        for (int movable : Bits.iterate(movablePegs(board))) {
            var holes = reachableHoles(board, Bits.bit(movable));
            for (int hole : Bits.iterate(holes))
                n += countSolutions(jump(board, movable, hole));
        }
        numSolutions.put(normalized, n);
        return n;
    }
}
```

Exercises

Exercise 6.5. Estimate how long it would take a modern computer to generate all 41 quadrillion solutions of peg solitaire.

Exercise 6.6. How many distinct boards does the call

```
countSolutions(PegBoard.INIT_PEGS);
```

encounter? What fraction of those boards are actually winnable?

Exercise 6.7. Write a program that uses memoization to compute the Fibonacci sequence

$$F_0 = F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2.$$

Exercise 6.8. The interface `Function<T, R>` defined in Java's standard library represents a general function from values of type `T` to values of type `R`. Write a class `Memoizer` that "memoizes" a given non-recursive `Function` by wrapping it and caching all return values. For example, the following listing creates a memoized version of the function `v -> v.toString()` that converts integers to their string representation:

```
var f = new Memoizer<Integer, String>(v -> v.toString());
assert f.apply(1).equals("1");
```

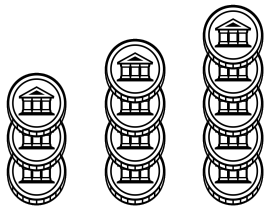
Bits
  bit() 5.5
  iterate() 5.9
Function →
Integer →
PegBoard 6.1
  FINAL_PEGS 6.1
  jump() 6.2
  movablePegs() 6.4
  reachableHoles() 6.4
PegCount
  countSolutions() 6.9
  normalize() 6.6
  numSolutions 6.9
String →

Any additional call to `apply(1)` should return the cached result.

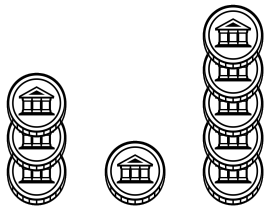
6.4 Problems

Nim

Exercise 6.9. ***Nim*** is a simple two-player game that is played with piles of coins or other small objects. A common version is 3,4,5-Nim in which there are three piles of 3, 4, and 5 coins:



The players alternately remove one or more coins from a single pile. For example, if the first player removes three coins from the second pile, we end up with the following position:



The player who removes the last coin wins.

What is a good strategy for playing Nim? More specifically, is there a ***winning strategy***, that is, a strategy that leads to a guaranteed win, regardless of how the opponent plays? Let's start our analysis with with a few simple instances of Nim. If there is a single pile of n coins, the first player obviously wins by taking all n coins, but taking fewer coins is risky because it gives the second player a chance to take all remaining coins. Taking all coins is therefore the only winning strategy for Nim with a single pile.

a) Study n, m-Nim in which there are two piles of n and m coins respectively. Try to find winning strategies for a few small values of n and m.

Since every move in Nim leads to a position with fewer coins, we can use recursion to determine whether there is a winning strategy for the current position. Let's define a function W that determines whether a position is winnable. Continuing

with the example of n, m-Nim, this strategy can be formulated as follows. If there are no coins left when our turn begins, we lose, so we set

$$W(0,0) = \mathsf{F}.$$

(We use the symbols T and F to represent *true* and *false*, respectively.) Other positions are winnable if at least one of the possible next moves leads to a position that is *not* winnable for the opponent. If the current state is (n, m), there are $n + m$ possible next moves, which lead to the following states:

$$(n-1,m), (n-2,m), \ldots, (0,m), (n,m-1), (n,m-2), \ldots, (n,0).$$

We can win the game if at least one of these states is not winnable (for the opponent), in other words if

$$\begin{aligned} W(n,m) = {} & \neg W(n-1,m) \vee \neg W(n-2,m) \vee \ldots \vee \neg W(0,m) \\ & \vee \neg W(n,m-1) \vee \neg W(n,m-2) \vee \ldots \vee \neg W(n,0) \end{aligned} \tag{6.2}$$

For example, Eq. (6.2) tells us that $(1,0)$ is a winning position because

$$W(1,0) = \neg W(0,0) = \neg \mathsf{F} = \mathsf{T}.$$

On the other hand, $(1,1)$ is not a winning position because

$$W(1,1) = \neg W(0,1) \vee \neg W(1,0) = \neg \mathsf{T} \vee \neg \mathsf{T} = \mathsf{F}.$$

Here we have used the fact that the stacks of coins are interchangeable, so $W(0,1) = W(1,0)$. As the number of coins and piles increases, the number of terms that must be considered grows rapidly.

b) Manually evaluate $W(1,1,2)$.

c) Implement a recursive function that determines whether a given Nim position is winnable. The function should work with any number of piles and coins. Is the position $10, 5, 4, 3$ winnable for the current player? What moves are possible? *Hint:* Use normalization and memoization.

The mathematical theory behind Nim was first studied by Charles L. Bouton in 1901. Bouton showed that there is a winning strategy for any number of piles and coins and derived a simple non-recursive procedure for determining the winner. His main idea was to write the number of coins in each pile as a binary number and add them *without carry*. The position is winning if and only if the result is nonzero. For example, when the number of coins is $(1,3,7)$, we obtain the sum

0	0	1	$= 1$
0	1	1	$= 3$
1	1	1	$= 7$
1	0	1	sum, ignoring carries

Since the result is 101 and not zero, $(1,3,7)$ must be a winning position. The binary sum without carries is also called the ***Nim sum*** and is equivalent to the XOR operation.

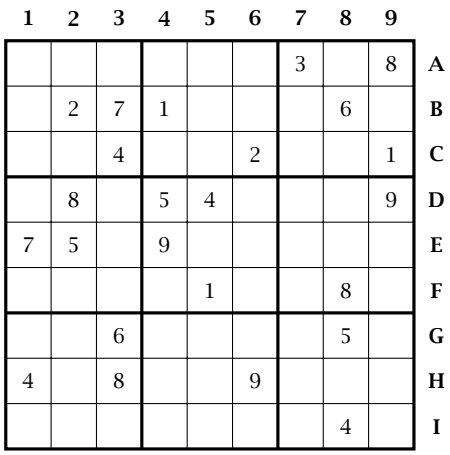


Figure 6.7 Example of a Sudoku puzzle.

d) Are $(2, 5, 6)$ and $(2, 5, 7)$ winning positions?

e) Write a program that uses Nim sums to find all winning moves for a given position. What are the winning moves for $(3, 6, 6)$?

Solving Sudoku

Exercise 6.10. ***Sudoku*** is a popular puzzle that is played on a 9×9 board. Some of the board's cells already contain digits. The objective is to fill the remaining cells with digits between 1 and 9 so that every row, column, and 3×3 block contains each digit exactly once. Sudoku puzzles are usually designed in such a way that there is exactly one solution that satisfies these conditions.

For the Sudoku board shown in Fig. 6.7, 24 of the 81 digits are given and the values of the remaining 57 digits must be found. Some numbers are easy to determine. For instance, it's not hard to see that the square labeled A8 must be 2: We know that this digit must occur somewhere in the top-right square, but rows B and C are blocked because there is already a 2 in squares B2 and C6, so the only remaining position is A8. Similar arguments can be used to successively fill in the remaining squares.

a) Manually solve the Sudoku board in Fig. 6.7.

Even though there are specialized algorithms for solving Sudoku and similar puzzles efficiently, modern computers are fast enough that a simple backtracking approach is viable as well.

b) Write a program that uses backtracking to solve a given Sudoku puzzle. *Hint:* Design a data structure that lets you keep track of the allowed digits in each cell.

6.5 Chapter Notes

Countless variations of ***peg solitaire*** can be obtained by changing the geometry of the board and the initial and final positions of the pegs [40, 11]. New and interesting questions also arise if we shift our focus from individual jumps to sequences of jumps. For instance, if we treat successive jumps with a single peg as a single move, how many such moves are necessary to clear the entire board? The answer is 18, and finding the corresponding solution is left as an exercise for the reader.

This chapter was inspired by an article by Bell [12]. There are also strategies for solving peg solitaire that do not rely on trial and error or brute computational force; for an overview, see the book by Berlekamp et al. [15, Chapter 23].

Problems that involve searching or analyzing arrangements of objects are called ***combinatorial problems***. Sometimes the goal is to find a certain specimen among all possible arrangements, for example when playing the lottery or trying to crack a combination lock. Other times, we consider the entirety of arrangements: How many different solutions does peg solitaire have, or what is the total number of different chess games?

Our discussion of peg solitaire highlighted an issue that plagues many combinatorial problems: Small instances of the game are easy to analyze, but as we increase the number of pegs, the number of possible solutions grows so quickly that it becomes impossible to generate them all using "brute force." This phenomenon is known as ***combinatorial explosion***, and it occurs in almost every situation in which we investigate arrangements of objects. Whenever there is more than one way to choose an object for each position in the arrangement, the total number of combinations grows at least exponentially, quickly outpacing the capabilities of even the most powerful computers.

The extraordinary speed of exponential growth is illustrated in Fig. 6.8. The lower four curves are growth rates that are commonly encountered in computer science. As the size of n increases along the horizontal axes, these functions show a moderate growth. (Both axes are logarithmic so the quadratic growth n^2 appears as a line.) In contrast, the growth of the exponential function 2^n is constantly accelerating and quickly reaches astronomical values.

We used ***bitboards*** to represent the different configurations of pegs and holes in peg solitaire. Similar techniques can be used in many other puzzles that involve the placement or movement of pieces on a board. A natural example is chess, where the 64 squares of the board naturally map to the 64 bits of a `long` integer. A slight complication in this case is that chess pieces come in two colors and six different types: kings, queens, rooks, bishops, knights, and pawns. If we use one bitboard for each type of piece, we can represent the complete state of a chessboard using twelve bitboards. Bitwise techniques can the then be used to quickly perform many important operations, such as computing possible moves or checking which pieces threaten other pieces. A popular open-source chess engine that uses bitboards is *Stockfish* (`https://stockfishchess.org/`). For a survey of bitboard techniques similar to the ones discussed in this chapter, see the article by Browne [21].

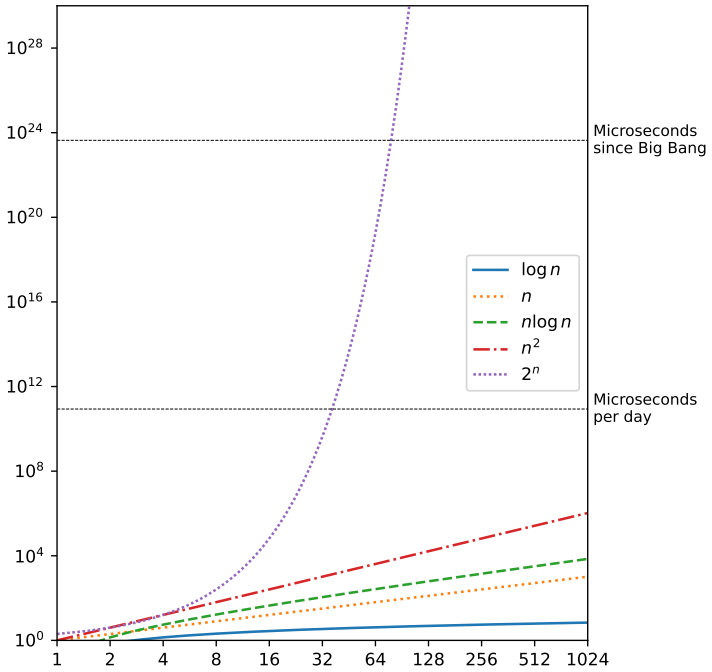


Figure 6.8 Comparison of growth rates. Exponential functions grow so quickly that they quickly reach astronomical values.

The game ***Nim***, which we discussed in Exercise 6.9 on page 143, was first described in 1901 by the American mathematician Charles L. Bouton [19]. Nim is one of the simplest examples of what mathematicians call a ***combinatorial game***, a two-player game that doesn't involve chance and in which both players have complete information about the state of the game. Other popular examples of combinatorial games are tic-tac-toe, Connect 4, checkers, and chess. For every combinatorial game, we can ask whether there is a ***winning strategy*** that guarantees a victory even if the opponent plays optimally. In the paper cited above, Bouton showed that there is an optimal strategy for playing Nim. The field of ***combinatorial game theory*** extends this analysis to a wide range of other games [3].

We saw that the winning strategy for Nim can be implemented using just a few XOR operations, which made it possible to construct Nim-playing machines several years before the development of electronic computers. The first such machine was the ***Nimatron***, a massive electro-mechanical device that was showcased at the World's Fair in New York in 1940, where it managed to win 90% of the 100 000 games it played [81].

The main technique we used to solve combinatorial problems in this chapter

was ***backtracking***, which tries all possible steps or moves until it happens upon a suitable solution. Modern computers are so fast that backtracking alone can solve most simple combinatorial puzzles, but if the number of possible moves grows too large, the number of potential solutions becomes prohibitive and more sophisticated algorithms must be tried to reduce the amount of trial-and-error required.

In the case of Sudoku, two approaches are especially noteworthy. Both convert each Sudoku puzzle into a ***constraint satisfaction problem*** (CSP), which consists of a set of variables whose values we wish to determine (in our case, the numbers in the squares of the Sudoku board) and a set of constraints those variables must satisfy (which include both the rules of the game and the existing numbers on the board).

One algorithmic technique for solving or at least simplifying such problems is known as ***constraint propagation*** [72]. This approach resembles the way humans solve Sudoku puzzles: If we know the number in a particular square, we can "propagate" this knowledge to constrain the range of allowed numbers of other squares in the same row, column, and block; we then repeatedly propagate these new constraints in the same way.

The second approach is to translate the problem into a ***Boolean formula*** and then search for a set of variables that *satisfies* this formula. Let's use the variable x_{ijk} to indicate that the square in row i and column j has value k and its negation $\bar{x}_{ijk}$ that the value is not k. With this convention, we can encode the initial state of the board and the general rules of the game using Boolean formulas. For example, the formula

$$x_{A33} \wedge x_{A98} \wedge x_{B22} \wedge x_{B37} \wedge x_{B41} \wedge x_{B86}$$

encodes the first two rows of the Sudoku board in Fig. 6.7 and the formula

$$x_{A1k} \wedge \bar{x}_{A2k} \wedge \bar{x}_{A3k} \wedge \bar{x}_{A4k} \wedge \bar{x}_{A5k} \wedge \bar{x}_{A6k} \wedge \bar{x}_{A7k} \wedge \bar{x}_{A8k} \wedge \bar{x}_{A9k}$$

states that if the value of the square A1 is k, no other square in row A can have that value. If we combine all these constraints into a single expression, we obtain a single large Boolean formula of the 9^3 variables x_{ijk}. Solving this formula means finding a subset of variables that, when set to *true*, make the whole formula *true*.

The general problem of solving Boolean formulas is known as the ***satisfiability problem*** (SAT) and belongs to the famous class of ***NP-hard problems***. Fortunately, many Boolean formulas that occur in practice (including those that describe Sudoku puzzles) are reasonably tame and can be solved efficiently using standard SAT solvers. An efficient Sudoku solver based on this idea can be found at `https://github.com/t-dillon/tdoku`. Combinatorial algorithms in general, including backtracking and SAT solvers, are discussed in detail in volume 4B of *The Art of Computer Programming* [62].

Bibliography

[1] T. Akenine-Möller et al. *Real-Time Rendering*. 4th ed. Boca Raton, FL, USA: A K Peters/CRC Press, 2018 (cit. on p. 86).

[2] J. Alber and R. Niedermeier. "On Multidimensional Curves with Hilbert Property". In: *Theory Comput. Systems* 33 (2000), pp. 295–312 (cit. on p. 86).

[3] M. H. Albert, R. J. Nowakowski, and D. Wolfe. *Lessons in Play. Introduction to Combinatorial Game Theory*. 2nd ed. CRC Press, 2019 (cit. on p. 147).

[4] K. Appel and W. Haken. "Every Planar Map is Four Colorable. Part I: Discharging". In: *Illinois J. Math.* 21(3) (1977), pp. 429–490 (cit. on p. 173).

[5] K. Appel, W. Haken, and J. Koch. "Every Planar Map is Four Colorable. Part II: Reducibility". In: *Illinois J. Math.* 21(3) (1977), pp. 491–567 (cit. on p. 173).

[6] J. Arndt and C. Haenel. *Pi Unleashed*. Springer, 2001 (cit. on p. 330).

[7] M. D. Atkinson. "The Cyclic Towers of Hanoi". In: *Information Processing Letters* 13(3) (Dec. 1981), pp. 118–119 (cit. on p. 63).

[8] M. Bader. *Space-Filling Curves. An Introduction with Applications in Scientific Computing*. Springer, 2013 (cit. on p. 86).

[9] D. H. Bailey et al. "The Quest for Pi". In: *Mathematical Intelligencer* 19(1) (Jan. 1997), pp. 50–56 (cit. on p. 330).

[10] M. Barnsley. *Fractals Everywhere*. 2nd ed. Academic Press, 2014 (cit. on p. 21).

[11] J. Beasley. *The Delights of Peg Solitaire*. 2021. URL: `http://www.jsbeasley.co.uk/delights.htm` (cit. on p. 146).

[12] G. I. Bell. *Notes on Solving and Playing Peg Solitaire on a Computer*. 2009. URL: `https://arxiv.org/abs/0903.3696` (cit. on p. 146).

[13] T. Bell and D. Kulp. "Longest-Match String Searching for Ziv-Lempel Compression". In: *Software: Practice and Experience* 23(7) (July 1993), pp. 757–771 (cit. on p. 273).

[14] A. Benjamin, G. Chartrand, and P. Zhang. *The Fascinating World of Graph Theory*. Princeton University Press, 2017 (cit. on p. 173).

[15] E. R. Berlekamp, J. R. Conway, and R. K. Guy. *Winning Ways for your Mathematical Plays*. 2nd ed. Vol. 4. CRC Press, 2004 (cit. on p. 146).

[16] J. F. Blinn. "What Is a Pixel?" In: *IEEE Comput. Graph. Appl.* 25(5) (Sept. 2005), pp. 82–87 (cit. on p. 44).

[17] J. Bloch. *Effective Java*. 3rd ed. Addison-Wesley Professional, Dec. 2017 (cit. on p. 339).

[18] P. Bourke. *Fractals, Chaos, Self-Similarity*. URL: `http://paulbourke.net/fractals/` (visited on 02/03/2026) (cit. on p. 43).

[19] C. L. Bouton. "Nim, a Game with a Complete Mathematical Theory". In: *Annals of Mathematics* 3(1/4) (1901), pp. 35–39 (cit. on p. 147).

[20] R. P. Brent and P. Zimmerman. *Modern Computer Arithmetic*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010 (cit. on p. 328).

[21] C. Browne. "Bitboard Methods for Games". In: *ICGA Journal* 37(2) (June 2014), pp. 67–84 (cit. on p. 146).

[22] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson, 2015 (cit. on pp. 122, 246).

[23] T. C. Chen and I. T. Ho. "Storage-Efficient Representation of Decimal Data". In: *CACM* 18(1) (Jan. 1975), pp. 49–52 (cit. on p. 436).

[24] S. A. Cook. "On the Minimum Computation Time of Functions". PhD thesis. Harvard University, Dept. of Mathematics, 1966 (cit. on p. 328).

[25] T. H. Cormen et al. *Introduction to Algorithms*. 4th ed. MIT Press, 2022 (cit. on pp. 173, 208, 293, 320).

[26] M. Cowlishaw. "Densely Packed Decimal Encoding". In: *IEE Proceedings – Computers and Digital Techniques* 149(3) (May 2002), pp. 102–104 (cit. on p. 436).

[27] J. Culberson. *Sokoban is PSPACE-Complete*. Tech. rep. TR97-02. University of Alberta, 1997 (cit. on p. 209).

[28] L. P. Deutsch. *DEFLATE Compressed Data Format Specification*. RFC 1951. Version 1.3. May 1996. URL: `https://www.ietf.org/rfc/rfc1951.txt` (cit. on p. 273).

[29] A. K. Dewdney. "Computer Recreations. A Computer Microscope Zooms In for a Look at the Most Complex Object in Mathematics". In: *Scientific American* (Aug. 1985), pp. 16–24 (cit. on p. 43).

[30] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271 (cit. on p. 208).

[31] S. Draves and E. Reckase. *The Fractal Flame Algorithm*. 2008. URL: `https://flam3.com/flame_draves.pdf` (visited on 02/03/2026) (cit. on p. 21).

[32] J. Duda et al. "The Use of Asymmetric Numeral Systems as an Accurate Replacement for Huffman coding". In: *2015 Picture Coding Symposium (PCS)*. 2015, pp. 65–69 (cit. on p. 236).

[33] L. Euler. “Solutio problematis ad geometriam situs pertinentis”. In: *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8 (1741), pp. 128–140 (cit. on p. 172).

[34] G. Farin. “A History of Curves and Surfaces in CAGD”. In: *Handbook of Computer Aided Geometric Design*. Ed. by G. Farin, J. Hoschek, and M.-S. Kim. Elsevier, 2002, pp. 1–22 (cit. on p. 86).

[35] G. Farin. *Curves and Surfaces for CAGD. A Practical Guide*. 5th ed. Morgan Kaufmann, 2002 (cit. on p. 86).

[36] F. A. Farris. *Creating Symmetry. The Artful Mathematics of Wallpaper Patterns*. Princeton, NJ, USA: Princeton University Press, 2015 (cit. on p. 84).

[37] G. Flegg. *Numbers: Their History and Meaning*. Dover, 2002 (cit. on p. 328).

[38] E. Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. on p. 183).

[39] M. Gardner. “Bouncing Balls in Polygons and Polyhedrons”. In: *Martin Gardner’s 6th Book of Mathematical Diversions*. University of Chicago Press, 1983. Chap. 4, pp. 29–38 (cit. on p. 174).

[40] M. Gardner. “Peg Solitaire”. In: *The Unexpected Hanging and Other Mathematical Diversions*. University of Chicago Press, 1991. Chap. 11, pp. 122–135 (cit. on pp. 125, 146).

[41] M. Gardner. *Penrose Tiles to Trapdoor Ciphers. …and the Return of Dr. Matrix*. Revised. The Mathematical Association of America, 1997 (cit. on p. 87).

[42] M. Gardner. “The Fantastic Combinations of John Conway’s New Solitaire Game “Life””. In: *Scientific American* 223(4) (Oct. 1970), pp. 120–123 (cit. on p. 44).

[43] C. Gotsman and M. Lindenbaum. “On the Metric Kroperties of Discrete Space-Filling Curves”. In: *IEEE Trans. Image Process.* 5(5) (May 1996), pp. 794–797 (cit. on p. 86).

[44] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. URL: `http://gmplib.org/` (cit. on p. 328).

[45] B. Grünbaum and G. C Shephard. *Tilings & Patterns*. 2nd ed. Dover, 2016 (cit. on p. 87).

[46] J. Hadley and D. Singmaster. “Problems to Sharpen the Young”. In: *The Mathematical Gazette* 76(475) (Mar. 1992), pp. 102–126 (cit. on p. 173).

[47] P. E. Hart, N. J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Trans. Syst. Sci. Cybern.* 4(2) (July 1968), pp. 100–107 (cit. on p. 209).

[48] D. Harvey and J. Van Der Hoeven. “Integer Multiplication in Time O(n log n)”. In: *Ann. of Math.* 193(2) (Mar. 2021), pp. 563–617 (cit. on p. 329).

[49] R. A. Hearn and E. D. Demaine. *Games, Puzzles, and Computation*. A K Peters, 2009 (cit. on p. 209).

[50] D. Hilbert. “Ueber die stetige Abbildung einer Linie auf ein Flächenstück”. In: *Mathematische Annalen* 38 (Sept. 1891), pp. 459–460 (cit. on p. 86).

[51] A. M. Hinz, S. Klavžar, and C. Petr. *The Tower of Hanoi – Myths and Maths*. 2nd ed. Cham: Birkhäuser, 2018 (cit. on p. 63).

[52] J. Hoppe. *Light Bulb Jokes*. 2012. URL: https://home.c-c-g.de/index.php/humor/120-light-bulb-jokes (visited on 02/04/2026) (cit. on p. 353).

[53] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40(9) (Sept. 1952), pp. 1098–1101 (cit. on p. 236).

[54] J. F. Hughes et al. *Computer Graphics. Principles and Practice*. 3rd ed. Addison-Wesley, 2013 (cit. on p. 63).

[55] N. Johnston and D. Green. *Conway's Game of Life. Mathematics and Construction*. 2022. URL: https://conwaylife.com/book/ (visited on 02/05/2026) (cit. on p. 44).

[56] A. Junghanns. "Pushing the Limits: New Developments in Single-Agent Search". PhD thesis. University of Alberta, 1999 (cit. on p. 209).

[57] A. Karatsuba and Y. Ofman. "Multiplication of Many-Digital Numbers by Automatic Computers". In: *Dokl. Akad. Nauk SSSR* 145(2) (1962), pp. 293–294 (cit. on p. 328).

[58] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall, 1988 (cit. on p. 381).

[59] E. Klarreich. "Multiplication Hits the Speed Limit". In: *Commun. ACM* 63(1) (Dec. 2019), pp. 11–13. URL: https://doi.org/10.1145/3371387 (cit. on p. 329).

[60] D. E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*. 3rd ed. Vol. 2. Addison-Wesley, 1997 (cit. on pp. 304, 328, 381).

[61] D. E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 1*. Vol. 4A. Addison-Wesley, 2011 (cit. on p. 122).

[62] D. E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 2*. Vol. 4B. Addison-Wesley, 2023 (cit. on p. 148).

[63] D. E. Knuth. *The Stanford GraphBase. A Platform for Combinatorial Computing*. Addison-Wesley, 1993 (cit. on p. 173).

[64] D. H. Larkin, S. Sen, and R. E. Tarjan. "A Back-to-Basics Empirical Study of Priority Queues". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Portland, Oregon: Society for Industrial and Applied Mathematics, 2014, pp. 61–72 (cit. on p. 208).

[65] A. Lempel and J. Ziv. "A Universal Algorithm for Sequential Data Compression". In: *IEEE Transactions on Information Theory* 23(3) (May 1977), pp. 337–343 (cit. on p. 273).

[66] A. Levitin and M. Levitin. *Algorithmic Puzzles*. Oxford University Press, 2011 (cit. on p. 173).

[67] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. URL: http://www.inference.org.uk/mackay/itila/ (visited on 02/05/2026) (cit. on pp. 235, 274).

[68] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1983 (cit. on p. 43).

[69] S. Marschner and P. Shirley. *Fundamentals of Computer Graphics*. 5th ed. A K Peters/CRC Press, 2021 (cit. on p. 63).

[70] A. Moffat. "Huffman Coding". In: *ACM Comput. Surv.* 52(4) (2019), 85:1–85:35. URL: https://doi.org/10.1145/3342555 (cit. on p. 236).

[71] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 (cit. on p. 328).

[72] P. Norvig. *Solving Every Sudoku Puzzle*. URL: http://www.norvig.com/sudoku.html (visited on 12/17/2022) (cit. on p. 148).

[73] Numberphile. *Why 381,654,729 is Awesome*. May 2013. URL: https://www.youtube.com/watch?v=gaVMrqzb91w (visited on 10/11/2021) (cit. on p. 452).

[74] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, 2001 (cit. on p. 328).

[75] G. Peano. "Sur une courbe, qui remplit toute une aire plane". In: *Mathematische Annalen* 36(1) (1890), pp. 157–160 (cit. on p. 86).

[76] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals. New Frontiers of Science*. 2nd ed. Springer, 2004 (cit. on pp. 21, 43).

[77] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. 4th ed. MIT Press, 2023. URL: https://www.pbr-book.org/ (visited on 02/05/2026) (cit. on p. 44).

[78] I. Pressman and D. Singmaster. ""The Jealous Husbands" and "The Missionaries and Cannibals"". In: *The Mathematical Gazette* 73(464) (June 1989), pp. 73–81 (cit. on pp. 173, 407).

[79] Quote Investigator. *Insanity Is Doing the Same Thing Over and Over Again and Expecting Different Results*. Mar. 2017. URL: https://quoteinvestigator.com/2017/03/23/same/ (visited on 10/11/2021) (cit. on p. 1).

[80] J. Rissanen and G. G. Langdon Jr. "Arithmetic Coding". In: *IBM J. Res. Develop.* 23(2) (Mar. 1979) (cit. on p. 236).

[81] L. Rougetet. "Machines Designed to Play Nim Games (1940–1970): A Possible (Re)Use in the Modern French Mathematics Curriculum?" In: *Teaching and Learning Discrete Mathematics Worldwide: Curriculum and Research*. Ed. by E. W. Hart and J. Sandefur. Cham: Springer International Publishing, 2018, pp. 229–250. URL: https://doi.org/10.1007/978-3-319-70308-4_15 (cit. on p. 147).

[82] K. Sadakane and H. Imai. "Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E83-A(12) (Dec. 2000), pp. 2689–2698 (cit. on p. 273).

[83] D. Salomon. *Data Compression: The Complete Reference*. 4th ed. With contributions by Giovanni Motta and David Bryant. Springer-Verlag, 2007 (cit. on p. 273).

[84] A. A. Sardinas and G. W. Patterson. “A Necessary and Sufficient Condition for Unique Decomposition of Coded Messages”. In: *Convention Record of the I.R.E., 1953 National Convention, Part 8: Information Theory* (1953), pp. 104–108 (cit. on p. 236).

[85] K. Sayood. *Introduction to Data Compression*. 5th ed. Morgan Kaufmann, 2018 (cit. on p. 273).

[86] P. Schneider and D. H. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 2002 (cit. on p. 63).

[87] R. Sedgewick and K. Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011 (cit. on pp. 173, 208, 225, 293).

[88] C. E. Shannon. “A Mathematical Theory of Communication”. In: 27(3) (July 1948), pp. 379–423 (cit. on p. 235).

[89] C. E. Shannon. “A Mathematical Theory of Communication”. In: 27(4) (Oct. 1948), pp. 623–656 (cit. on p. 235).

[90] M. Sipser. *Introduction to the Theory of Computation*. 3rd ed. Cengage Learning, 2013. Boston, MA (cit. on p. 209).

[91] J. A. Storer and T. G. Szymanski. “Data Compression via Textual Substitution”. In: *Journal of the ACM* 29(4) (Oct. 1982), pp. 928–951 (cit. on p. 273).

[92] *The Sokoban Wiki*. 2024. URL: `http://sokobano.de/wiki/index.php?title=Main_Page` (visited on 02/09/2026) (cit. on p. 209).

[93] A. L. Toom. “The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers”. Russian. In: *Dokl. Akad. Nauk SSSR* 150 (1963), pp. 496–498 (cit. on p. 328).

[94] M. C. K. Tweedie. “A Graphical Method of Solving Tartaglian Measuring Puzzles”. In: *The Mathematical Gazette* 23(255) (July 1939), pp. 278–282 (cit. on p. 174).

[95] H. S. Warren Jr. *Hacker’s Delight*. 2nd ed. Addison-Wesley, 2013 (cit. on p. 122).

[96] P. Wegner. “A Technique for Counting Ones in a Binary Computer”. In: *Communications of the ACM* 3(5) (May 1960), p. 322 (cit. on p. 382).

[97] T. A. Welch. “A Technique for High-Performance Data Compression”. In: *Computer* 17(6) (June 1984), pp. 8–19 (cit. on p. 273).

[98] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002. URL: `https://www.wolframscience.com/nks/` (cit. on p. 44).

[99] A. J. Yee. *y-cruncher - A Multi-Threaded Pi-Program*. URL: `http://www.numberworld.org/y-cruncher/` (visited on 03/09/2026) (cit. on p. 330).

[100] J. Ziv and A. Lempel. “Compression of Individual Sequences via Variable-Rate Coding”. In: *IEEE Transactions on Information Theory* 24(5) (Sept. 1978), pp. 530–536 (cit. on p. 273).

Index of Identifiers

A

abs(), in BigInt, 314, **315**
abs(), in Duration, **455**, 458
AbstractCollection, 118
add(), in BitSet, 118, 182, **385**, 418
addAll(), in BitSet, 119, 184, **388**
addDigits(), in Duration, 456, **457**, 458
addWord(), in SimilarWords, 205, **422**
advance(), in GameOfLife, 362, **363**
advanceInput(), in LZTokenizer, 250, 255, **256**
Affine, **11**, 11–13, 15, 51, 53–58, 359
  Affine(), **11**, 13, 53, 55
  compose(), **55**, 55, 56
  IDENTITY, **53**, 55, 58
  rotation(), **53**, 53, 55, 56
  scaling(), **53**, 53, 55, 56
  times(), **55**, 55, 58
  transformPoint(), **12**, 12, 15
  transformPoints(), **12**, 12, 54, 57, 58
  translation(), **53**, 53, 55, 56
Affine(), in Affine, **11**, 13, 53, 55
AffineIFS, 12, **13**, 13–15, 359
  AffineIFS(), 12, **13**, 13
  BARNSLEY_FERN, **13**
  chooseIndex(), 14, **15**
  extendedChaosGame(), 14, **15**
  probabilities, 12, **13**
  transforms, 12, **13**
AffineIFS(), in AffineIFS, 12, **13**, 13
ALL_DIRECTIONS, in Board, **179**, 188, 196, 420, 427
allCratesOnGoals(), in GameState, 182, **183**, 183, 190, 203
angle(), in Point, 18, **356**
appendDigit(), in BigNat, **307**, 308
arctan1overX(), in PiMachin, **459**, 459
ArithmeticException, 106, 316, 451, 452, 457
ArrayDeque, 157, 158, 167, 254, 413, 415
ArrayList, 6, 12, 15, 58, 77, 112, 113, 115, 154, 161, 166, 188, 196, 254, 337, 343–345, 350, 351, 365, 371, 378, 397, 401, 409, 411, 418, 420–422, 425, 427, 440
Arrays, 279, 343, 385, 386, 395, 400, 405, 406, 456
Ascii, 428, 429
  encryptCaesar(), 428, **429**
  isDigit(), **428**
  isLower(), **428**
  isUpper(), **428**
  toLower(), **428**
  toUpper(), **428**

B

BARNSLEY_FERN, in AffineIFS, **13**
BASE, in BigNat, **282**, 282
BasicPath, **165**, 165–168, 185, 200, 203, 401
  length(), **166**, 166, 203
  node(), **165**, 165, 166
  predecessor(), **165**, 165, 166
  printNodes(), 167, 168, **401**, 401
  toNodeList(), **166**, 166, 185, 203, 401
BezierCurve, **72**, 72, 75–77, 371, 373
  closestPoint(), **373**
  draw(), **77**
  isStraight(), 76, **77**, 373
  length(), **371**
  mid(), **75**
  pointAt(), **72**, 72, 373
  split(), **75**, 75, 77, 371, 373
  subdivide(), 76, **77**, 77

BigInt, 275, **314**, 314–319, 450, 453, 459
  abs(), 314, **315**
  BigInt(), **314**, 314, 315
  compareTo(), 316, 319, **450**
  divideAndRemainder(), **319**
  dividedBy(), **319**, 459
  equals(), 314, **315**, 453, 459
  hashCode(), 314, **315**
  minus(), **318**, 459
  negate(), 314, **315**, 318, 459
  ONE, **315**
  plus(), **318**, 318, 453
  remainder(), **319**, 453
  signum(), 314, **315**
  TEN, **315**, 453
  times(), **319**, 453, 459
  toLong(), **316**, 316, 317
  valueOf(), **315**, 315–317, 453, 459
  ZERO, **315**, 318, 453, 459
BigInt(), in BigInt, **314**, 314, 315
BigInteger, 328
BigNat, 275, **278**, 278–286, 288, 290–292, 296–298, 301, 303, 306–308, 310–316, 318, 319, 322, 446–450, 452
  appendDigit(), **307**, 308
  BASE, **282**, 282
  BigNat(), **279**, 283, 285, 288, 291, 296–298, 303, 307, 447
  compareTo(), 281, **282**, 306, 307, 318, 450
  digit(), **278**, 278, 282, 288, 291, 303, 307, 447
  DIGIT_BITS, 281, **282**, 283, 285, 288, 291, 447
  DIGIT_MASK, 281, **282**, 283, 285, 288, 291, 447
  digitOr0(), **278**, 278, 283, 285, 288, 296, 307, 446
  divideAndRemainder(), **306**, 306, 319, 449
  divideByDigit(), 301, **303**, 306, 307
  dividedBy(), **306**, 306
  divideLong(), 306, **307**
  divideLongInternal(), 306, **307**
  equals(), 279, **280**, 291, 306, 314, 315, 449
  fromDecimal(), 308, 311, **312**
  fromUnsignedLong(), 282, **283**, 283, 303, 312, 315, 447
  hashCode(), 279, **280**, 315
  longFromDecimal(), 310, **311**, 311, 312, 448
  minus(), **288**, 290, 298, 307, 318
  multiplyBasePower(), 297, **298**, 298
  multiplySimple(), **291**, 297, 298
  ONE, **283**, 452
  plus(), 284, **285**, 286, 290, 298, 310, 312, 318
  pow(), 312, 322, 447, 449, **452**
  remainder(), **306**, 306
  slice(), 296, **297**, 297, 298, 307, 447
  TEN, **283**, 312, 447, 449
  times(), 297, **298**, 310, 312, 319, 447, 452
  timesDigit(), 307, **447**
  toString(), 313, 448, **449**
  toUnsignedLong(), **283**, 283, 316, 449
  ZERO, **283**, 283, 306, 314
BigNat(), in BigNat, **279**, 283, 285, 288, 291, 296–298, 303, 307, 447
bit(), in Bits, 107, **108**, 127, 132, 136, 140, 142, 399, 406, 407
BitCount, 382, 390
  bitCountKernighan(), **382**
  bitCountSubdivide(), **390**
bitCountKernighan(), in BitCount, **382**
bitCountSubdivide(), in BitCount, **390**
BitInputStream, 219–221, 223, 229–232, 259, 263, **434**, 434, 435, 441, 442
  BitInputStream(), **434**
  fillBuffer(), **434**, 434, 435
  readBit(), 220, 223, 259, **435**
  readBits(), 220, 221, 223, 232, 259, 263, 434, **435**, 441, 442
BitInputStream(), in BitInputStream, **434**
BitIterator, 113, **114**, 114, 115, 387
  BitIterator(), **114**, 115
  hasNext(), 113, **114**
  next(), 113, **114**
BitIterator(), in BitIterator, **114**, 115
bitLength(), in IntUtil, **222**, 222, 262, 263, 265, 266
BitmapFont, 90, 92, 93, 379
  digitsImage, **90**, 92, 93
  drawNumber(), **93**, 93
  printImage(), **92**
  writePbm(), **379**
BitOutputStream, 219, 220, 222, 228, 231, 232, 258, 262, 263, 267, 432, **433**, 433, 438
  BitOutputStream(), **433**
  flush(), 232, 432, **433**
  writeBit(), 220, 222, 232, 258, **433**
  writeBits(), 220, 222, 232, 258, 262, 263, 267, 432, **433**
BitOutputStream(), in BitOutputStream, **433**

Bits, 107–111, 114, 115, 127, 128, 132, 136, 140, 142, 391, 399, 400, 405–407
  bit(), 107, **108**, 127, 132, 136, 140, 142, 399, 406, 407
  bits(), 107, **108**, 109, 127, 128, 132
  contains(), 110, **111**, 407
  containsAll(), 110, **111**
  containsNone(), 110, **111**
  containsSome(), 110, **111**
  countTrailingZeros(), **391**
  iterate(), 114, **115**, 115, 132, 136, 140, 142, 400, 405–407
  range(), **108**, 108, 127, 405, 407
  toList(), **115**, 132
bits(), in Bits, 107, **108**, 109, 127, 128, 132
BitSet, 117, **118**, 118, 119, 181–185, 189, 190, 193, 385–389, 417, 418, 420
  add(), 118, 182, **385**, 418
  addAll(), 119, 184, **388**
  BitSet(), 181, 184, 193, **386**, 417, 418, 420
  contains(), 118, 385, **386**, 417, 418
  containsAll(), 119, 387, **388**
  equals(), 118, 119, **387**, 387
  hashCode(), 118, 119, **387**
  isEmpty(), 119, 387, **388**
  iterator(), 119, 193, **389**
  remove(), 118, 385, **386**
  removeAll(), 119, 193, **388**
  retainAll(), 119, **388**
  size(), 119, 387, **388**
BitSet(), in BitSet, 181, 184, 193, **386**, 417, 418, 420
BitSet.Iter, **389**, 389
  hasNext(), **389**
  Iter(), **389**
  next(), **389**
  remove(), **389**
blend(), in IFSBlend, **359**
blocked(), in GameState, **189**, 190
BLOCKER, in LaserPiece, 422, **423**
blockIndex(), in Sudoku, **399**
Board, 177, **178**, 178–182, 185, 188–190, 192, 193, 196, 417, 420, 423–427
  ALL_DIRECTIONS, **179**, 188, 196, 420, 427
  Board(), **178**
  DOWN, **179**, 179, 424
  getDirection(), **179**, 180, 185, 192, 193
  getX(), 177, **178**, 179, 190, 417
  getY(), 177, **178**, 179, 190, 417
  height(), **178**, 417
  index(), 177, **178**, 182, 185, 190, 417
  isInside(), **178**, 179
  LEFT, **179**, 179, 424
  moveIndex(), **179**, 179, 180, 188, 189, 193, 196, 420
  RIGHT, **179**, 179, 424
  UP, **179**, 179, 424
  width(), **178**, 417
Board(), in Board, **178**
board, in LaserBoard, **425**
Board.Direction, **179**, 179, 185, 192, 193, 421, 423–425, 427
  charCode(), **179**, 185, 192, 424
  xOff(), **179**, 179, 425
  yOff(), **179**, 179, 425
BOARD_MASK, in Connect4, **394**, 394
Boolean, 397
bottomLeft(), in Rectangle, 32, 37, **355**
BOUNDARY_LEFT, in PegBoard, **128**
BOUNDARY_RIGHT, in PegBoard, **128**
boundingBox(), in Rectangle, 354, **355**
branchL, in PythagorasTree, **56**, 57
branchR, in PythagorasTree, **56**, 57
Buddhabrot, **36**, 36–39, 360, 361
  Buddhabrot(), **36**, 36, 39
  computeHistogram(), **38**, 39, 360, 361
  drawBuddhabrot(), **39**
  drawNebulabrot(), **361**
  followTrajectory(), **37**, 38
  imageHeight, **36**
  imageWidth, **36**
  maximum(), **39**, 361
  region, **36**
Buddhabrot(), in Buddhabrot, **36**, 36, 39
BufferedImage, 19, 32, 39, 356, 357, 361, 412, 413, 415
BufferedReader, 421
buildSquares(), in PythagorasTree, **58**, 58
buildTree(), in PythagorasTree, **58**, 58
buildTreeSimple(), in PythagorasTree, **57**, 57
Byte, 229, 334
  toUnsignedInt(), 229
BYTE_TAG, in LZSS, **257**, 258

C

causesDeadlock(), in GameState, 189, **190**

ceilBinaryLog(), in IntMath, **106**, 106, 248
CellName, 313, 448, **450**, 450
  CellName(), **450**
  column(), **450**
  fromString(), 448, **450**
  row(), **450**
  toString(), 448, **450**
CellName(), in CellName, **450**
Cellular1D, 382, 385
  evolveCells(), **385**
  nextState(), 382, **385**
ChaosGame, 6, 7
  chaosGame(), **6**, 6, 7
  drawSierpinski(), 6, **7**
chaosGame(), in ChaosGame, **6**, 6, 7
Character, 93, 192, 311, 343, 427
charCode(), in Board.Direction, **179**, 185, 192, 424
checkTrip(), in JealousCouples, **406**, 406
checkWin(), in Connect4, **394**, 394
chooseIndex(), in AffineIFS, 14, **15**
CivBoard, 426, **427**, 427
  CivBoard(), **427**
  edges(), **427**
  getTerrain(), **427**
  Pos, **427**
CivBoard(), in CivBoard, **427**
CivBoard.Pos, 427
closestParam(), in LineSegment, 372, **373**, 373
closestPoint(), in BezierCurve, **373**
closestPoint(), in LineSegment, 372, **373**
Codeword, in PrefixCode, 220, 430
Collection, 154, 155, 343, 386, 406, 421–423
Collections, 166, 427
  reverse(), 166
Color, 360
column(), in CellName, **450**
Comparable, 227, 278, 281, 314
Comparator, 201
compare(), in Integer, 281, 282, 450
compareTo(), in BigInt, 316, 319, **450**
compareTo(), in BigNat, 281, **282**, 306, 307, 318, 450
compareTo(), in Duration, 455, **456**, 458
compareTo(), in HuffmanCode.Entry, **227**
Components, 418
  findComponents(), **418**
compose(), in Affine, **55**, 55, 56
compress(), in HuffmanPacker, **229**, 229
compress(), in LZHEncoder, 266, **267**
compress(), in LZSSEncoder, **258**, 258, 259
compressMoves(), in SokobanSolver, 203, 418, **419**
computeCodewords(), in PrefixCode, 218, **219**, 219
computeHistogram(), in Buddhabrot, **38**, 39, 360, 361
computePi(), in PiMachin, **459**, 459
computeSteps(), in SokobanSolver, **191**, 191, 203
Connect4, 394
  BOARD_MASK, **394**, 394
  checkWin(), **394**, 394
  fourBitsSet(), **394**, 394
  HEIGHT, **394**, 394
  WIDTH, **394**, 394
construct(), in HuffmanCode, **228**, 265
Consumer, 158, 159, 161, 350
contains(), in Bits, 110, **111**, 407
contains(), in BitSet, 118, 385, **386**, 417, 418
contains(), in Rectangle, 354, **355**
containsAll(), in Bits, 110, **111**
containsAll(), in BitSet, 119, 387, **388**
containsNone(), in Bits, 110, **111**
containsSome(), in Bits, 110, **111**
countLeadingZeros(), in LeadingZeros, **392**
countSolutions(), in PegCount, 141, **142**, 142
countTrailingZeros(), in Bits, **391**
crates(), in GameState, **183**, 188, 193, 196
crates(), in Level, **181**, 185
createFromTable(), in PrefixCodeFromTable, **430**, 430
Curve, 376, **377**, 377, 378
  pointAt(), 376, **377**, 378
  tMax(), 376, **377**, 378
  tMin(), 376, **377**, 378
CurvePlotter, 377, **378**, 378
  draw(), 377, **378**
  subdivide(), 377, **378**
CyclicBuffer, 247, **248**, 248–250, 255, 256, 441, 442
  CyclicBuffer(), **248**, 248, 249, 441
  get(), 247, **248**, 250, 255, 442
  set(), 247, **248**, 249, 256, 442
CyclicBuffer(), in CyclicBuffer, **248**, 248, 249, 441

D

DecimalCoder, 435, 437, 438

writeBCD(), 435, **437, 438**
writeDeclets(), **438**
decode(), in LZ4Decoder, **444**
decodeBlock(), in LZHDecoder, 441, **442**
DecodeException, **444**, 444, 445
decodeInteger(), in LZHDecoder, **442**
decodeLength(), in LZ4Decoder, 444, **445**, 445
decodeLiteral(), in LZ4Decoder, 444, **445**
decodeMatch(), in LZ4Decoder, 444, **445**
decodeSymbol(), in PrefixCode, 220, **221**, 230, 442
decompress(), in HuffmanPacker, **230**, 230
decompress(), in LZHDecoder, **441**, 441
decompress(), in LZSSDecoder, 258, **259**
deflateThick(), in Penrose, 375, **376**
deflateThin(), in Penrose, 375, **376**
Deque, 254, 256, 343, 344
diameter(), in GraphUtil, **163**, 163
dictionary, in LZHDecoder, **441**
DieHardGraph, **153**, 153–156, 159, 160, 162, 167
DieHardGraph(), **153**, 156, 159, 160, 162, 167
makeNode(), **153**, 153–155
neighbors(), 153, 154, **155**, 156
nodes(), 153, **154**
DieHardGraph(), in DieHardGraph, **153**, 156, 159, 160, 162, 167
digit(), in BigNat, **278**, 278, 282, 288, 291, 303, 307, 447
DIGIT_BITS, in BigNat, 281, **282**, 283, 285, 288, 291, 447
DIGIT_MASK, in BigNat, 281, **282**, 283, 285, 288, 291, 447
digitOr0(), in BigNat, **278**, 278, 283, 285, 288, 296, 307, 446
digitsImage, in BitmapFont, **90**, 92, 93
Dijkstra, **201**, 201–203
Dijkstra(), **201**, 201, 203
nextNode(), 201, **202**, 202, 203
shortestPath(), 202, **203**, 203
traverse(), 202, **203**
Dijkstra(), in Dijkstra, **201**, 201, 203
distance(), in GraphLayer, **160**, 163, 196
distance(), in Point, **354**, 371, 373
divideAndRemainder(), in BigInt, **319**
divideAndRemainder(), in BigNat, **306**, 306, 319, 449
divideByDigit(), in BigNat, 301, **303**, 306, 307
dividedBy(), in BigInt, **319**, 459
dividedBy(), in BigNat, **306**, 306
divideLong(), in BigNat, 306, **307**
divideLongInternal(), in BigNat, 306, **307**
Double, 334
down(), in PegBoard, **128**, 128, 130, 131
DOWN, in Board, **179**, 179, 424
draw(), in BezierCurve, **77**
draw(), in CurvePlotter, 377, **378**
drawBuddhabrot(), in Buddhabrot, **39**
drawMandelbrot(), in MandelbrotPainter, 31, **32**, 33
drawNebulabrot(), in Buddhabrot, **361**
drawNumber(), in BitmapFont, **93**, 93
drawPoints(), in IFSPainter, **357**
drawSierpinski(), in ChaosGame, 6, **7**
Duration, **455**, 455–458
abs(), **455**, 458
addDigits(), 456, **457**, 458
compareTo(), 455, **456**, 458
Duration(), **455**, 457
equals(), **456**
hashCode(), **456**
makeSeconds(), 456, **457**, 458
minus(), 456, **458**
negate(), **455**, 458
ofUnit(), **457**
plus(), 456, **458**
seconds(), 456, **457**
subDigits(), 456, **457**, 458
total(), **458**
totalWeeks(), 456, **458**
Duration(), in Duration, **455**, 457

E

EAST, in HilbertCurve, 68, **69**, 371
eccentricity(), in GraphUtil, **163**, 163
Edge(), in WeightedGraph.Edge, **195**
edges(), in CivBoard, **427**
edges(), in WeightedGraph, 194, **195**, 202
edges(), in WeightedPushGraph, **196**, 196
Ellipse2D, 7
emit(), in LaserPiece, 421, **423**, 425
Emitter, **423**
EMPTY, in LaserPiece, 422, **423**, 425, 426
EMPTY, in PrefixCodeFromTable, **430**
encryptCaesar(), in Ascii, 428, **429**
end(), in JealousCouples, **407**
ensure(), in LZ4Decoder, **444**, 445
Entry, 228

Entry(), in HuffmanCode.Entry, **227**, 228
Entry, in HuffmanCode, 227
equals(), in BigInt, 314, **315**, 453, 459
equals(), in BigNat, 279, **280**, 291, 306, 314, 315, 449
equals(), in BitSet, 118, 119, **387**, 387
equals(), in Duration, **456**
escapeTime(), in JuliaSet, 360, **362**
escapeTime(), in MandelbrotSet, **31**, 32, 38
everyone(), in JealousCouples, **407**
evolveCells(), in Cellular1D, **385**
Exception, 444
extendedChaosGame(), in AffineIFS, 14, **15**

F

farthestLayer(), in GraphUtil, 162, **163**, 163, 164
fillBuffer(), in BitInputStream, **434**, 434, 435
FINAL_PEGS, in PegBoard, **127**, 136, 142
findComponents(), in Components, **418**
findMatches(), in LZTokenizer, 250, 255, **256**
findPathTo(), in Graph, **167**, 167, 185, 190, 192
findSimilarWords(), in SimilarWords, 205, **422**, 423
FLIP_H, in PegCount, 138, **139**
flipDigit(), in WolfGoatCabbageGraph, **403**
flipH(), in PegCount, 138, **139**, 139
Float, 334, 336, 343
FloodFill, 413, 415
  floodFill_pixelBased, **413**
  floodFill_spanBased(), **415**
FloodFill.PixelPos, **413**, 415
floodFill_pixelBased, in FloodFill, **413**
floodFill_spanBased(), in FloodFill, **415**
floorBinaryLog(), in IntMath, **106**, 106
flush(), in BitOutputStream, 232, 432, **433**
followTrajectory(), in Buddhabrot, **37**, 38
fourBitsSet(), in Connect4, **394**, 394
fromDecimal(), in BigNat, 308, 311, **312**
fromDigits(), in Radix, **449**
fromString(), in CellName, 448, **450**
fromUnsignedLong(), in BigNat, 282, **283**, 283, 303, 312, 315, 447
front, in LZTokenizer, **249**, 250, 255, 256
FULL_BOARD, in PegBoard, **127**, 129, 131
Function, 142, 396

G

GameOfLife, 362, **363**, 363
  advance(), 362, **363**
  GameOfLife(), **363**
  get(), 362, **363**
  set(), 362, **363**
GameOfLife(), in GameOfLife, **363**
GameState, 182, **183**, 183, 185, 188–193, 196, 203, 416, 417
  allCratesOnGoals(), 182, **183**, 183, 190, 203
  blocked(), **189**, 190
  causesDeadlock(), 189, **190**
  crates(), **183**, 188, 193, 196
  GameState(), **183**, 203
  toString(), 416, **417**
  tryPush(), 188, **189**, 189, 196
  workerPos(), **183**, 188, 192, 193, 196
GameState(), in GameState, **183**, 203
get(), in CyclicBuffer, 247, **248**, 250, 255, 442
get(), in GameOfLife, 362, **363**
getByte(), in LZToken, **245**, 258, 265, 266
getDirection(), in Board, **179**, 180, 185, 192, 193
getLength(), in LZToken, **245**, 245, 258, 265, 266
getOffset(), in LZToken, **245**, 245, 258, 265, 266
getTerrain(), in CivBoard, **427**
getX(), in Board, 177, **178**, 179, 190, 417
getY(), in Board, 177, **178**, 179, 190, 417
goals(), in Level, **181**, 183, 190, 417, 420
Graph, **153**, 153–155, 157–163, 166–168, 183–185, 188, 190, 192, 194, 196, 403, 405, 409, 411, 417, 418, 420, 423–426
  findPathTo(), **167**, 167, 185, 190, 192
  neighbors(), **153**, 153, 155, 158, 161, 167, 194
  nodes(), **153**, 153, 154
  scan(), 157, **158**, 158–160, 417
  scanLayers(), 160, **161**, 161, 196
  scanPaths(), 166, **167**, 167
  visitLayers(), 160, **161**, 162, 163, 196
  visitNodes(), 158, **159**, 159, 188, 418, 420, 425, 426
Graphics2D, 6, 7, 356
GraphLayer, **160**, 160, 161, 163, 196
  distance(), **160**, 163, 196
  GraphLayer(), **160**, 161
  nodes(), **160**, 196
  start(), **160**
GraphLayer(), in GraphLayer, **160**, 161
GraphUtil, 162–164
  diameter(), **163**, 163
  eccentricity(), **163**, 163

farthestLayer(), 162, **163**, 163, 164
radius(), **163**, 163
greeting, in Hello, **xi**, xi

H

HanoiGraph, 410, **411**, 411
neighbors(), 410, **411**
hashCode(), in BigInt, 314, **315**
hashCode(), in BigNat, 279, **280**, 315
hashCode(), in BitSet, 118, 119, **387**
hashCode(), in Duration, **456**
hashFront(), in LZTokenizer, **255**, 255, 256
HashMap, 152, 201, 202, 227, 339, 343, 345, 396, 397
HashSet, 155, 157, 158, 161, 167, 345, 406, 422, 423, 426
hasNext(), in BitIterator, 113, **114**
hasNext(), in BitSet.Iter, **389**
hasRemaining(), in LZTokenizer, **249**, 251, 258, 267
height(), in Board, **178**, 417
height(), in Rectangle, 32, 354, **355**
HEIGHT, in Connect4, **394**, 394
Hello, **xi**, xi
greeting, **xi**, xi
printGreeting(), **xi**
HilbertCurve, 68, **69**, 69, 70, 370, 371
EAST, 68, **69**, 371
hilbertCurve(), 69, **70**, 70, 370, 371
NORTH, 68, **69**, 70, 371
printPoints(), **371**
SOUTH, 68, **69**, 371
turn(), 68, **69**, 70
WEST, 68, **69**, 371
hilbertCurve(), in HilbertCurve, 69, **70**, 70, 370, 371
HuffmanCode, 227, 228, 265
construct(), **228**, 265
Entry, 227
HuffmanCode.Entry, **227**, 227, 228
compareTo(), **227**
Entry(), **227**, 228
HuffmanPacker, 229, 230
compress(), **229**, 229
decompress(), **230**, 230

I

IDENTITY, in Affine, **53**, 55, 58
IFSBlend, 359
blend(), **359**
lerp(), **359**
IFSPainter, **357**, 357, 358
drawPoints(), **357**
IFSPainter(), **358**
IFSPainter(), in IFSPainter, **358**
IllegalArgumentException, 267, 297, 385
imageHeight, in Buddhabrot, **36**
imageWidth, in Buddhabrot, **36**
index(), in Board, 177, **178**, 182, 185, 190, 417
index, in LZTokenizer, **254**, 256
INDEX_SIZE, in LZTokenizer, **254**, 255
INIT_PEGS, in PegBoard, **127**, 136, 141
initBuffers(), in LZTokenizer, 248, **249**, 249
initIndex(), in LZTokenizer, 248, 249, **254**, 254
input, in LZHDecoder, **441**
input, in LZTokenizer, **249**
InputStream, 228, 229, 232, 248, 249, 256, 258, 267, 434, 438
insert(), in PrefixCodeFromTable, **430**, 430
Integer, 46, 47, 106, 113–115, 118, 120, 142, 155, 157, 161, 183, 184, 219, 222, 227, 281, 282, 334, 336, 343, 350, 351, 389, 397, 405, 409, 427, 450, 456, 457
compare(), 281, 282, 450
numberOfLeadingZeros(), 222
intersection(), in Rectangle, 354, **355**
IntMath, 106, 248, 451
ceilBinaryLog(), **106**, 106, 248
floorBinaryLog(), **106**, 106
pow(), **451**, 451
IntUtil, 222, 262, 263, 265, 266
bitLength(), **222**, 222, 262, 263, 265, 266
IOException, 249, 421, 433–435, 437, 438
isByteToken(), in LZToken, **245**, 245, 258, 265, 266
isDigit(), in Ascii, **428**
isEmpty(), in BitSet, 119, 387, **388**
isEmpty(), in Rectangle, 354, **355**
isInside(), in Board, **178**, 179
isLevelClosed(), in LevelClosed, **417**
isLower(), in Ascii, **428**
isStraight(), in BezierCurve, 76, **77**, 373
isStraight(), in QuadraticBezier, **374**, 374
isUpper(), in Ascii, **428**
isWinning(), in NimMemoize, 396, **397**
Iter, 119
Iter(), in BitSet.Iter, **389**
Iterable, 114, 115, 119, 153, 185, 343–345, 409

iterate(), in Bits, 114, **115**, 115, 132, 136, 140, 142, 400, 405–407
Iterator, 113, 114, 119, 344, 345, 387, 389
iterator(), in BitSet, 119, 193, **389**

J

JealousCouples, **405**, 405–407
  checkTrip(), **406**, 406
  end(), **407**
  everyone(), **407**
  makeState(), 406, **407**, 407
  MAX_COUPLES, **405**
  MEN_MASK, **405**
  noJealousy(), **405**, 406
  normalize(), **407**
  start(), **407**
  WOMEN_MASK, **405**
JealousCouples.State, 404, **405**, 405–407
JuliaSet, 360, 362
  escapeTime(), 360, **362**
jump(), in PegBoard, **127**, 136, 142

L

LaserBoard, 424, **425**, 425, 426
  board, **425**
  neighbors(), **425**
  Pos, **425**, 425, 426
  Ray, **425**, 425, 426
  simulateLight(), **426**
LaserPiece, 421, 422, **423**, 423–426
  BLOCKER, 422, **423**
  emit(), 421, **423**, 425
  EMPTY, 422, **423**, 425, 426
  reflect(), 421, 422, **423**, 425
LeadingZeros, 392
  countLeadingZeros(), **392**
left(), in PegBoard, 127, **128**, 128, 130, 131
LEFT, in Board, **179**, 179, 424
length(), in BasicPath, **166**, 166, 203
length(), in BezierCurve, **371**
LENGTH_BITS, in LZSS, **257**
lerp(), in IFSBlend, **359**
lerp(), in Point, **6**, 6, 75, 365, 372–374, 376
Level, **181**, 181–185, 188–190, 196, 203, 416–418, 420
  crates(), **181**, 185
  goals(), **181**, 183, 190, 417, 420
  Level(), **181**
  parse(), 181, **416**, 416
  resetSquare(), **182**
  setCode(), 181, **182**, 416
  walls(), **181**, 417, 420
  workerStart(), **181**, 417, 418, 420
Level(), in Level, **181**
level, in MoveGraph, **184**
level, in PushGraph, **188**
LevelClosed, 417
  isLevelClosed(), **417**
LineSegment, 372, 373
  closestParam(), 372, **373**, 373
  closestPoint(), 372, **373**
  LineSegment(), **373**, 373
  pointAt(), 372, **373**
LineSegment(), in LineSegment, **373**, 373
LinkedList, 343
List, 7, 12, 15, 56, 58, 70, 77, 115, 136, 152, 158, 161, 166, 167, 185, 188, 190, 191, 196, 254, 343, 344, 350, 351, 365, 374, 376, 378, 397, 403, 409, 411, 418, 420, 423–425, 427, 440
  of(), 158
Long, 112–114, 120, 315–317, 387–389
  MAX_VALUE, 316
  MIN_VALUE, 316
longFromDecimal(), in BigNat, 310, **311**, 311, 312, 448
lookaheadSize, in LZTokenizer, **249**, 250, 255, 256
LZ4Decoder, **272**, 272, 444, 445
  decode(), **444**
  decodeLength(), 444, **445**, 445
  decodeLiteral(), 444, **445**
  decodeMatch(), 444, **445**
  ensure(), **444**, 445
LZHDecoder, **441**, 441, 442
  decodeBlock(), 441, **442**
  decodeInteger(), **442**
  decompress(), **441**, 441
  dictionary, **441**
  input, **441**
  LZHDecoder(), **441**
  out, **441**
  outLength, **441**
  write(), 441, **442**
LZHDecoder(), in LZHDecoder, **441**
LZHEncoder, 263, **264**, 264–268, 439
  compress(), 266, **267**
  LZHEncoder(), **267**

maxLength, **264**, 265–267
maxOffset, **264**, 266, 267
minLength, **264**, 265–267
offsetCode, **264**, 264, 265, 267
prepareCodes(), **265**, 266, 267
tagCode, **264**, 264, 265, 267
writeToken(), **266**, 266, 267
LZHEncoder(), in LZHEncoder, **267**
LZSS, **257**, 257–259
BYTE_TAG, **257**, 258
LENGTH_BITS, **257**
MATCH_TAG, **257**, 258
MAX_LENGTH, **257**, 258
MAX_OFFSET, **257**, 258
MIN_LENGTH, **257**, 258
OFFSET_BITS, **257**, 258
LZSSDecoder, 258, 259
decompress(), 258, **259**
LZSSEncoder, 258, 259
compress(), **258**, 258, 259
LZToken, 245, 250, 258, 265, 266
getByte(), **245**, 258, 265, 266
getLength(), **245**, 245, 258, 265, 266
getOffset(), **245**, 245, 258, 265, 266
isByteToken(), **245**, 245, 258, 265, 266
makeByte(), **245**, 245, 250
makeMatch(), **245**, 245, 250
LZTokenizer, 248, **249**, 249–251, 254–256, 258, 267
advanceInput(), 250, 255, **256**
findMatches(), 250, 255, **256**
front, **249**, 250, 255, 256
hashFront(), **255**, 255, 256
hasRemaining(), **249**, 251, 258, 267
index, **254**, 256
INDEX_SIZE, **254**, 255
initBuffers(), 248, **249**, 249
initIndex(), 248, 249, **254**, 254
input, **249**
lookaheadSize, **249**, 250, 255, 256
LZTokenizer(), 248, **249**, 258, 267
maxLength, **249**, 249, 250
maxOffset, **249**, 249, 256
minLength, **249**, 250, 255
nextToken(), **250**, 251, 258
readTokens(), 250, **251**, 267
window, **249**, 250, 255, 256
LZTokenizer(), in LZTokenizer, 248, **249**, 258, 267

M

makeByte(), in LZToken, **245**, 245, 250
makeInner(), in PrefixCode, **218**, 218, 223, 228, 430
makeLeaf(), in PrefixCode, **218**, 218, 223, 228, 430
makeMatch(), in LZToken, **245**, 245, 250
makeNode(), in DieHardGraph, **153**, 153–155
makeSeconds(), in Duration, 456, **457**, 458
makeState(), in JealousCouples, 406, **407**, 407
MandelbrotPainter, 31–33, 36
drawMandelbrot(), 31, **32**, 33
zoomRect(), **33**, 33, 36
MandelbrotSet, 31, 32, 38
escapeTime(), **31**, 32, 38
Map, 201, 343, 344, 397, 424
MATCH_TAG, in LZSS, **257**, 258
Math, 7, 38, 53, 69, 72, 139, 155, 219, 255, 296, 297, 307, 317, 334, 335, 354, 356, 360, 361, 363, 368, 373, 416, 440, 457
min(), 139, 296
MAX_COUPLES, in JealousCouples, **405**
MAX_LENGTH, in LZSS, **257**, 258
MAX_OFFSET, in LZSS, **257**, 258
MAX_VALUE, in Long, 316
maximum(), in Buddhabrot, **39**, 361
maxLength, in LZHEncoder, **264**, 265–267
maxLength, in LZTokenizer, **249**, 249, 250
maxOffset, in LZHEncoder, **264**, 266, 267
maxOffset, in LZTokenizer, **249**, 249, 256
maxSymbol(), in PrefixCode, 218, **219**
Memoizer, 395
MEN_MASK, in JealousCouples, **405**
mid(), in BezierCurve, **75**
min(), in Math, 139, 296
MIN_LENGTH, in LZSS, **257**, 258
MIN_VALUE, in Long, 316
minLength, in LZHEncoder, **264**, 265–267
minLength, in LZTokenizer, **249**, 250, 255
minus(), in BigInt, **318**, 459
minus(), in BigNat, **288**, 290, 298, 307, 318
minus(), in Duration, 456, **458**
movablePegs(), in PegBoard, 130, **131**, 132, 136, 142
MoveGraph, 183, **184**, 184–186, 188, 192, 196, 417, 418, 420
level, **184**
MoveGraph(), 183, **184**, 185, 188, 192, 196, 417, 418, 420

neighbors(), 183, **184**, 185
obstacles, **184**
MoveGraph(), in MoveGraph, 183, **184**, 185, 188, 192, 196, 417, 418, 420
moveIndex(), in Board, **179**, 179, 180, 188, 189, 193, 196, 420
multiplyBasePower(), in BigNat, 297, **298**, 298
multiplySimple(), in BigNat, **291**, 297, 298
MysteryCurve, **377**, 377
pointAt(), **377**
tMax(), **377**
tMin(), **377**

N

NavigableMap, 344
NavigableSet, 344
negate(), in BigInt, 314, **315**, 318, 459
negate(), in Duration, **455**, 458
neighbors(), in DieHardGraph, 153, 154, **155**, 156
neighbors(), in Graph, **153**, 153, 155, 158, 161, 167, 194
neighbors(), in HanoiGraph, 410, **411**
neighbors(), in LaserBoard, **425**
neighbors(), in MoveGraph, 183, **184**, 185
neighbors(), in PushGraph, **188**, 188, 193
neighbors(), in ThreeJugProblem, 408, **409**
neighbors(), in WolfGoatCabbageGraph, **403**
neighbors(), in WordLadderGraph, **423**
nested(), in Whirl, **365**
next(), in BitIterator, 113, **114**
next(), in BitSet.Iter, **389**
nextNode(), in Dijkstra, 201, **202**, 202, 203
nextState(), in Cellular1D, 382, **385**
nextToken(), in LZTokenizer, **250**, 251, 258
Nim, 398
printWinningMoves(), **398**, 398
NimMemoize, 396, **397**, 397
isWinning(), 396, **397**
NimMemoize(), 396, **397**
normalize(), 396, **397**
NimMemoize(), in NimMemoize, 396, **397**
node(), in BasicPath, **165**, 165, 166
node(), in Path, 165, **166**, 167
node(), in WeightedPath, **200**, 202
Node, in PrefixCode, 221, 222, 227, 430
nodes(), in DieHardGraph, 153, **154**
nodes(), in Graph, **153**, 153, 154
nodes(), in GraphLayer, **160**, 196
nodes(), in ThreeJugProblem, 408, **409**
nodes(), in WolfGoatCabbageGraph, **403**
nodes(), in WordLadderGraph, **423**
noJealousy(), in JealousCouples, **405**, 406
normalize(), in JealousCouples, **407**
normalize(), in NimMemoize, 396, **397**
normalize(), in PegCount, 138, **139**, 142
NORTH, in HilbertCurve, 68, **69**, 70, 371
NumberFormatException, 449
numberOfLeadingZeros(), in Integer, 222
numSolutions, in PegCount, **142**, 395

O

Object, 163, 334, 337, 339, 342, 343
Objects, 339
obstacles, in MoveGraph, **184**
of(), in List, 158
OFFSET_BITS, in LZSS, **257**, 258
offsetCode, in LZHEncoder, **264**, 264, 265, 267
ofUnit(), in Duration, **457**
ONE, in BigInt, **315**
ONE, in BigNat, **283**, 452
out, in LZHDecoder, **441**
outLength, in LZHDecoder, **441**
OutputStream, 230, 232, 259, 433, 437, 441

P

parse(), in Level, 181, **416**, 416
parseDigit(), in Radix, 448, **449**, 449
Path, 165, **166**, 166–168
node(), 165, **166**, 167
Path(), **166**, 167, 168
predecessor(), 165, **166**
Path(), in Path, **166**, 167, 168
PegBoard, 126, **127**, 127–132, 136, 141, 142
BOUNDARY_LEFT, **128**
BOUNDARY_RIGHT, **128**
down(), **128**, 128, 130, 131
FINAL_PEGS, **127**, 136, 142
FULL_BOARD, **127**, 129, 131
INIT_PEGS, **127**, 136, 141
jump(), **127**, 136, 142
left(), 127, **128**, 128, 130, 131
movablePegs(), 130, **131**, 132, 136, 142
reachableHoles(), 130, **131**, 132, 136, 142
reachablePositions(), 130, **131**
right(), **128**, 128–131
up(), **128**, 128, 130, 131

PegCount, 138–142, 395
countSolutions(), 141, **142**, 142
FLIP_H, 138, **139**
flipH(), 138, **139**, 139
normalize(), 138, **139**, 142
numSolutions, **142**, 395
ROTATE, 138, **139**
rotate(), 138, **139**, 139
shuffleBits(), 138, 139, **140**
PegSolve, 136
printSolutions(), **136**, 136
Penrose, 375, 376
deflateThick(), 375, **376**
deflateThin(), 375, **376**
PiMachin, 459
arctan1overX(), **459**, 459
computePi(), **459**, 459
planMovement(), in SokobanSolver, 191, **192**, 192
plus(), in BigInt, **318**, 318, 453
plus(), in BigNat, 284, **285**, 286, 290, 298, 310, 312, 318
plus(), in Duration, 456, **458**
Point, 4, **5**, 5–7, 12, 15, 18, 33, 37, 54, 56, 58, 72, 75, 77, 354, 356, 365, 368, 371–374, 376, 378
angle(), 18, **356**
distance(), **354**, 371, 373
lerp(), **6**, 6, 75, 365, 372–374, 376
Point(), **5**, 6, 7, 33, 54, 56, 356, 368, 371, 372
polar(), **356**
radius(), 18, **356**
x(), **5**, 33
y(), **5**, 33
Point(), in Point, **5**, 6, 7, 33, 54, 56, 356, 368, 371, 372
pointAt(), in BezierCurve, **72**, 72, 373
pointAt(), in Curve, 376, **377**, 378
pointAt(), in LineSegment, 372, **373**
pointAt(), in MysteryCurve, **377**
polar(), in Point, **356**
Pos, in CivBoard, **427**
Pos, in LaserBoard, **425**, 425, 426
pow(), in BigNat, 312, 322, 447, 449, **452**
pow(), in IntMath, **451**, 451
predecessor(), in BasicPath, **165**, 165, 166
predecessor(), in Path, 165, **166**
predecessor(), in WeightedPath, **200**
Predicate, 157, 158, 161, 167, 202, 203, 351
PrefixCode, 217, **218**, 218–223, 227–230, 263, 266, 267, 430, 442
Codeword, 220, 430
computeCodewords(), 218, **219**, 219
decodeSymbol(), 220, **221**, 230, 442
makeInner(), **218**, 218, 223, 228, 430
makeLeaf(), **218**, 218, 223, 228, 430
maxSymbol(), 218, **219**
Node, 221, 222, 227, 430
PrefixCode(), 218, **219**, 223, 228
readCode(), **223**, 230, 442
readTree(), 222, **223**, 223
writeCode(), **222**, 222, 229, 267
writeSymbol(), **220**, 220, 229, 263, 266, 267
writeTree(), 221, **222**, 222
PrefixCode(), in PrefixCode, 218, **219**, 223, 228
PrefixCode.Codeword, 217, **218**, 219
PrefixCode.Node, 217, **218**, 218, 219, 223, 228, 430
PrefixCodeFromTable, 430
createFromTable(), **430**, 430
EMPTY, **430**
insert(), **430**, 430
prepareCodes(), in LZHEncoder, **265**, 266, 267
printGreeting(), in Hello, **xi**
printImage(), in BitmapFont, **92**
printNodes(), in BasicPath, 167, 168, **401**, 401
printPoints(), in HilbertCurve, **371**
printSolutions(), in PegSolve, **136**, 136
printWinningMoves(), in Nim, **398**, 398
PrintWriter, 379
PriorityQueue, 201, 202, 208, 228
probabilities, in AffineIFS, 12, **13**
PushGraph, **188**, 188, 190, 193
level, **188**
neighbors(), **188**, 188, 193
PushGraph(), **188**, 190
PushGraph(), in PushGraph, **188**, 190
putDigit(), in Sudoku, 398, **399**, 400
PythagorasTree, **56**, 56–58
branchL, **56**, 57
branchR, **56**, 57
buildSquares(), **58**, 58
buildTree(), **58**, 58
buildTreeSimple(), **57**, 57
PythagorasTree(), **56**
UNIT_SQUARE, **56**, 57, 58
PythagorasTree(), in PythagorasTree, **56**

Q

QuadraticBezier, **374**, 374
  isStraight(), **374**, 374
  subdivide(), **374**
Queue, 344

R

radius(), in GraphUtil, **163**, 163
radius(), in Point, 18, **356**
Radix, 448, 449
  fromDigits(), **449**
  parseDigit(), 448, **449**, 449
Random, 6, 350, 351
range(), in Bits, **108**, 108, 127, 405, 407
Ray, in LaserBoard, **425**, 425, 426
reachableHoles(), in PegBoard, 130, **131**, 132, 136, 142
reachablePositions(), in PegBoard, 130, **131**
readBit(), in BitInputStream, 220, 223, 259, **435**
readBits(), in BitInputStream, 220, 221, 223, 232, 259, 263, 434, **435**, 441, 442
readCode(), in PrefixCode, **223**, 230, 442
readTokens(), in LZTokenizer, 250, **251**, 267
readTree(), in PrefixCode, 222, **223**, 223
readWords(), in WordList, 420, **421**
Rectangle, 16, 31–33, 36, 37, 354, **355**, 355
  bottomLeft(), 32, 37, **355**
  boundingBox(), 354, **355**
  contains(), 354, **355**
  height(), 32, 354, **355**
  intersection(), 354, **355**
  isEmpty(), 354, **355**
  Rectangle(), 33, **355**
  topRight(), **355**
  width(), 32, 37, 354, **355**
Rectangle(), in Rectangle, 33, **355**
reflect(), in LaserPiece, 421, 422, **423**, 425
reflect(), in TransparentMirror, 422, **424**
region, in Buddhabrot, **36**
remainder(), in BigInt, **319**, 453
remainder(), in BigNat, **306**, 306
remove(), in BitSet, 118, 385, **386**
remove(), in BitSet.Iter, **389**
removeAll(), in BitSet, 119, 193, **388**
removeDigit(), in Sudoku, 398, **399**, 400
resetSquare(), in Level, **182**
retainAll(), in BitSet, 119, **388**
reverse(), in Collections, 166
reverse(), in ReverseList, **365**
ReverseList, 365
  reverse(), **365**
right(), in PegBoard, **128**, 128–131
RIGHT, in Board, **179**, 179, 424
rotate(), in PegCount, 138, **139**, 139
ROTATE, in PegCount, 138, **139**
rotation(), in Affine, **53**, 53, 55, 56
row(), in CellName, **450**
RuntimeException, 311, 427

S

scaling(), in Affine, **53**, 53, 55, 56
scan(), in Graph, 157, **158**, 158–160, 417
scanLayers(), in Graph, 160, **161**, 161, 196
scanPaths(), in Graph, 166, **167**, 167
seconds(), in Duration, 456, **457**
Set, 343, 344, 423, 426
set(), in CyclicBuffer, 247, **248**, 249, 256, 442
set(), in GameOfLife, 362, **363**
setCode(), in Level, 181, **182**, 416
shortestPath(), in Dijkstra, 202, **203**, 203
shuffleBits(), in PegCount, 138, 139, **140**
signum(), in BigInt, 314, **315**
SimilarWords, 205, **206**, 206, 422, 423
  addWord(), 205, **422**
  findSimilarWords(), 205, **422**, 423
  SimilarWords(), **206**, 423
SimilarWords(), in SimilarWords, **206**, 423
SimpleToken, **243**, 244, 246, 440
SimpleToken.Byte, **243**, 440
SimpleToken.Match, **243**, 440
simulateLight(), in LaserBoard, **426**
size(), in BitSet, 119, 387, **388**
slice(), in BigNat, 296, **297**, 297, 298, 307, 447
SokobanSolver, 191–193, 203, 418, 419
  compressMoves(), 203, 418, **419**
  computeSteps(), **191**, 191, 203
  planMovement(), 191, **192**, 192
  workerBeforePush(), 192, **193**
solve(), in Sudoku, **400**, 400
SOUTH, in HilbertCurve, 68, **69**, 371
split(), in BezierCurve, **75**, 75, 77, 371, 373
start(), in GraphLayer, **160**
start(), in JealousCouples, **407**
String, xi, 93, 142, 153–155, 157, 159, 167, 181, 204, 205, 254, 308, 311, 312, 334, 342, 343,

379, 401, 403, 411, 416, 417, 419, 421–423, 449
StringBuilder, 191, 192, 400, 417, 419, 422, 450
Strings, 400
subDigits(), in Duration, 456, **457**, 458
subdivide(), in BezierCurve, 76, **77**, 77
subdivide(), in CurvePlotter, 377, **378**
subdivide(), in QuadraticBezier, **374**
Sudoku, 398, **399**, 399, 400
  blockIndex(), **399**
  putDigit(), 398, **399**, 400
  removeDigit(), 398, **399**, 400
  solve(), **400**, 400
  Sudoku(), **399**, 400
Sudoku(), in Sudoku, **399**, 400
Supplier, 350
System, xi, 92, 132, 136, 159, 167, 168, 272, 298, 307, 334, 350, 368, 371, 398, 400, 444, 445

T

tagCode, in LZHEncoder, **264**, 264, 265, 267
target(), in WeightedGraph.Edge, **195**, 202
TEN, in BigInt, **315**, 453
TEN, in BigNat, **283**, 312, 447, 449
ThreeJugProblem, 408, 409
  neighbors(), 408, **409**
  nodes(), 408, **409**
times(), in Affine, **55**, 55, 58
times(), in BigInt, **319**, 453, 459
times(), in BigNat, 297, **298**, 310, 312, 319, 447, 452
timesDigit(), in BigNat, 307, **447**
tMax(), in Curve, 376, **377**, 378
tMax(), in MysteryCurve, **377**
tMin(), in Curve, 376, **377**, 378
tMin(), in MysteryCurve, **377**
toList(), in Bits, **115**, 132
toLong(), in BigInt, **316**, 316, 317
toLower(), in Ascii, **428**
toNodeList(), in BasicPath, **166**, 166, 185, 203, 401
topRight(), in Rectangle, **355**
toString(), in BigNat, 313, 448, **449**
toString(), in CellName, 448, **450**
toString(), in GameState, 416, **417**
total(), in Duration, **458**
totalWeeks(), in Duration, 456, **458**
totalWeight(), in WeightedPath, **200**, 201–203
toUnsignedInt(), in Byte, 229
toUnsignedLong(), in BigNat, **283**, 283, 316, 449
toUpper(), in Ascii, **428**
transformPoint(), in Affine, **12**, 12, 15
transformPoints(), in Affine, **12**, 12, 54, 57, 58
transforms, in AffineIFS, 12, **13**
translation(), in Affine, **53**, 53, 55, 56
TransparentMirror, 422, **424**, 424
  reflect(), 422, **424**
traverse(), in Dijkstra, 202, **203**
tryPush(), in GameState, 188, **189**, 189, 196
turn(), in HilbertCurve, 68, **69**, 70

U

UNIT_SQUARE, in PythagorasTree, **56**, 57, 58
up(), in PegBoard, **128**, 128, 130, 131
UP, in Board, **179**, 179, 424

V

valueOf(), in BigInt, **315**, 315–317, 453, 459
visitLayers(), in Graph, 160, **161**, 162, 163, 196
visitNodes(), in Graph, 158, **159**, 159, 188, 418, 420, 425, 426

W

walls(), in Level, **181**, 417, 420
weight(), in WeightedGraph.Edge, **195**, 202
WeightedGraph, 194, **195**, 195, 196, 201–203, 426, 427
  edges(), 194, **195**, 202
WeightedGraph.Edge, **195**, 195, 196, 202, 427
  Edge(), **195**
  target(), **195**, 202
  weight(), **195**, 202
WeightedPath, **200**, 200–203
  node(), **200**, 202
  predecessor(), **200**
  totalWeight(), **200**, 201–203
  WeightedPath(), **200**, 202
WeightedPath(), in WeightedPath, **200**, 202
WeightedPushGraph, **196**, 196, 203
  edges(), **196**, 196
  WeightedPushGraph(), **196**, 203
WeightedPushGraph(), in WeightedPushGraph, **196**, 203
WEST, in HilbertCurve, 68, **69**, 371
Whirl, 365
  nested(), **365**

whirl(), **365**
whirl(), in Whirl, **365**
width(), in Board, **178**, 417
width(), in Rectangle, 32, 37, 354, **355**
WIDTH, in Connect4, **394**, 394
window, in LZTokenizer, **249**, 250, 255, 256
WolfGoatCabbageGraph, **403**, 403
flipDigit(), **403**
neighbors(), **403**
nodes(), **403**
WOMEN_MASK, in JealousCouples, **405**
WordLadderGraph, **423**, 423
neighbors(), **423**
nodes(), **423**
WordLadderGraph(), **423**
WordLadderGraph(), in WordLadderGraph, **423**
WordList, 420, 421
readWords(), 420, **421**
workerBeforePush(), in SokobanSolver, 192, **193**
workerPos(), in GameState, **183**, 188, 192, 193, 196
workerStart(), in Level, **181**, 417, 418, 420
write(), in LZHDecoder, 441, **442**
writeBCD(), in DecimalCoder, 435, **437, 438**
writeBit(), in BitOutputStream, 220, 222, 232, 258, **433**
writeBits(), in BitOutputStream, 220, 222, 232, 258, 262, 263, 267, 432, **433**
writeCode(), in PrefixCode, **222**, 222, 229, 267
writeDeclets(), in DecimalCoder, **438**
writePbm(), in BitmapFont, **379**
writeSymbol(), in PrefixCode, **220**, 220, 229, 263, 266, 267
writeToken(), in LZHEncoder, **266**, 266, 267
writeTree(), in PrefixCode, 221, **222**, 222

X

x(), in Point, **5**, 33
xOff(), in Board.Direction, **179**, 179, 425

Y

y(), in Point, **5**, 33
yOff(), in Board.Direction, **179**, 179, 425

Z

ZERO, in BigInt, **315**, 318, 453, 459
ZERO, in BigNat, **283**, 283, 306, 314
zoomRect(), in MandelbrotPainter, **33**, 33, 36

Index

A

A* algorithm, 208, 209
access modifiers, 335, 338, 341
accessor methods, 165, 338
adaptive subdivision, 75–78, 85
ADD, 284, 288, 289, 291
addition, 283–285
adjacency lists, 152, 153
affine iterated function systems, 358
  blending, 19–21
  chaos game, 13
  choosing probabilities, 17
  examples, 20
  implementation, 12
affine transformations, 9, 12, 14, 56, 59, 63
  blending, 19
  composing, 53–54, 58
  elementary, 51–53
  implementation, 10–12, 53–55
aliasing, 44
Alice in Wonderland, 149, 260, 269
ambiguous codes, 214, 217
AND operator, 92, 109, 247, 288
animation of Julia sets, 39
anonymous classes, 347
anonymous functions, *see* lambda expressions
anonymous objects, 162, 349
arbitrary precision integers, *see* big integers
arbitrary-precision fractions, 327
Archimedes' recurrence formula, 325
arctan, power series, 325
arithmetic coding, 236
arithmetic shift, 101, 102, 288
ASCII, 212, 213, 216, 227, 428
  case conversion, 216
  control characters, 214
  relation to Unicode, 232
  relation to UTF-8, 233
aspect ratio, 33
asymmetric numeral systems, 236
Avogadro constant, 292
axis-aligned rectangles, 16

B

backtracking, 133–137, 145, 148, 400
Barnsley fern, 8–10, 12, 17, 19, 20, 22
base, *see* radix
BCD, *see* binary-coded decimals
Bézier curves, 86
  coordinate transformation, 81
  cubic, 70–73
  length of, 78
  parametric form, 71
  quadratic, 80–81
  recursive subdivision, 74–78
  straightness, 75
BFS, *see* breadth-first search
big integers
  addition, 284–285
  comparison, 279, 281
  conversion, 282, 283
  division, 301–308
  hashing, 279
  implementations of, 328
  multiplication, 290–292, 296–298
  representation, 276–279, 281–283
  signed, 314–319
  subtraction, 286–288

three-way comparison, 281
big-endian order, 277
big-Oh notation, 293
big-Omega notation, 293
big-Theta notation, 293
bignums, *see* big integers
binary-coded decimals, 234
binary codes, 212–217
binary heaps, 208
binary logarithm, 105, 106, 248
binary numbers, 90, 91, 95, 100, 113, 276
binary search, 121, 293
binary trees, 215–219, 223, 227, 228, 231
inserting into, 430
representation, 217
traversing, 218, 221
bit count, *see* population count
bit length, 221, 261–263, 268, 441, 451
bit masks, 92–93, 121, 281, 282, 393, 394
bit representation of numbers, *see* two's-complement representation
bit rotation, 94
bit sets, 112, 177, 180–183, 398, 404, 407
complement, 110
constructing, 107–109
large, 117–119
membership tests, 110
overview, 107
set operations, 109–112
bit shifts, 92–94, 99–101, 107–109, 127, 128, 248, 385, 394
bit streams, 228, 230
implementation, 231–232
padding, 229, 232
bit tricks, 122
bit width, 97, 221
bitboards, 126–129, 133, 138, 146, 180
bitlen function, *see* bit length
bitmap images, 90–94, 122
bits
iterating over, 92, 112–115
reading, 90–93
blinker life form, 41
Boolean formula, 148
borrow term, 286–288, 456
bounding box, 16
boxing of primitive data types, 337
breadth-first search, 151, 162, 164, 166, 170, 185, 188, 193, 200, 417, 425
implementation, 157–158
memory consumption, 186
overview, 156–157
relation to Dijkstra's algorithm, 197–198
visiting all layers, 160
visiting all nodes, 158, 159
BreadthFirstSearch, 157
Buddhabrot, 35, 36, 38
byte order, 277
bzip2 program, 268

C

Caesar cipher, 216
call stack, 48, 62, 63
carriage return, 214
carry term, 284–286, 288
Cartesian coordinates, 4, 18, 37
ceiling operator, 102, 103, 106, 331
cellular automata, 44, 117
elementary, 116–117
center of a graph, 162, 163
chaos game
for iterated function systems, 7–8, 13
implementation, 4–6, 14
overview, 2–4
theory of, 21–22
variations, 7, 17–19
visualizing point sets, 6–7, 19
ChaosGame, 3, 4
Chen-Ho encoding, 436
chess, use of bitboards, 146
ChooseIndex, 13–15
Chudnovsky formula, 330
Chudnovsky, David, 330
Chudnovsky, Gregory, 330
Civilization, computer game, 207
classes in Java, 333
code point in Unicode, 233
codewords, 211–215, 217–219, 229
ambiguous, 214
representation, 217
coding theory, 235, 274
collection classes, 343–344
collision in a hash table, 254
combinatorial explosion, 146
combinatorial games, 147

combinatorial problems, 146
comparators, 201, 352
complement, *see* set operations
complement in a positional number system, 287
complex numbers, 24–25, 30, 33
complexity classes, 209
complexity theory, 209
components of a graph, 186
compressing large files, 229, 266
compression algorithms, 268, 273
  see also Huffman coding, LZ4, LZ77, LZH, LZSS
computer arithmetic, 327, 328
  see also big integers, integer overflow, sign-magnitude representation, two's-complement representation
computer graphics, 70
computer-aided design, 86
Connect 4, 133
constraint propagation, 148
constraint satisfaction problem, 148
consumers, 349–350
contraction factor, 22
contractive functions, 21, 22
control characters, 214
control points, 71, 72, 81
convex hull, 72
cosine transform, 273
covariant return types, 185
cyclic buffers, 246–248, 441

D

Daft Punk, 237
data classes, 338, 339
data compression, 212, 235, 272
de Casteljau's algorithm, 74–76, 78, 80, 81
decanting problems, *see* water measurement problems
decimal expansion, 276, 322, 323
decimal numbers, 276, 277
decision problems, 209
declets, 235
Deflate algorithm, 273
deflation of Penrose tilings, 82
densely packed decimal encoding, 436
depth first search
  see also backtracking
depth of a node, 223
depth-first search, 134
design patterns, 183
  see also flyweight design pattern, iterators
determinants, 17
diameter of a graph, 162, 163, 169
dictionary region, 238, 248, 251, 255, 267
  indexing, 251–254
  searching, 250
  typical size, 251
dictionary-based compression, 237
die-hard graph, 159, 162, 169
  implementation, 153–154
  interior nodes, 156
  layers, 160
  overview, 151–152
  properties, 163–164
  visualization, 152
die-hard problem
  see also water measuring problems
  algorithmic solution, 167
  manual solution, 150
  overview, 149–151
  variations, 156, 160, 164
DIJKSTRA, 198
Dijkstra's algorithm, 197, 202
  generalization, 208
  implementation, 200–202
  overview, 197–200
  performance, 208
  visualization, 199
directed edges, 151
distance
  between points, 7
  between points and geometric objects, 79–80
distributive laws, 130, 132
DIVIDE, 300, 301, 304, 308, 448
divide-and-conquer method, 45–48, 120, 294, 320
DIVIDE-DIGIT, 301, 303, 306, 308
DIVIDE-LONG, 304–306, 308
divisibility, 322
division, 298–302
  by zero, 306
  of big integers, 301–308
  of signed integers, 318–319
double-ended queue, 254, 343, 344
dragon curve, 86
durations, computing with, 323–324

E

eccentricity of a node, 162–164
Eddington's number, 277
edges in a graph, 151
elementary cellular automata, 116, 117
Elias delta code, 274
Elias gamma code, 268, 274
emoticons, 233, 234
endianness, 277
entropy, 235
entropy codes, 235, 236, 273
equilateral triangle, 6
escape radius, 26–28, 31, 35, 36
escape time, 26, 28, 30, 31, 39, 360
EscapeTime, 30
Euler's theorem, 172
Eulerian cycle, 172
ExtendedChaosGame, 14

F

factorial function, 140
fast Fourier transform, 328, 329
Fibonacci heaps, 208
Fibonacci numbers, 59–61, 142
Fibonacci trees, 59–61
finite state entropy, 273
first-order functions, 346
fixed-length codes, 212–214
fixed-point numbers, 327
fixed-width integers, 275, 327
floating-point numbers, 327, 328
flood fill, 171–172
floor division, 101, 102, 331
floor operator, 101, 103, 331
flyweight design pattern, 183
four color problem, 173
Fourier transform, 273
fractal flame algorithm, 21
fractals, 2, 8, 13, 39, 42, 43, 78
  see also Hilbert curves, iterated function systems, Julia sets, Koch snowflage, Mandelbrot set
fractions, 327
From-Radix, 309–311, 450
From-Radix', 309, 310
functional interfaces, 158, 346, 347, 349–352
functional programming, 345–352
functional programming languages, 63

G

Game of Life, 40–42, 44, 116
generics, 337, 342–344
geometric mean, 325
geometric transformations, 50, 51, 53
  see also affine transformations
getter methods, 338
GIF file format, 273
glider gun, 42
glider life form, 41
golden ratio, 83
Gosper curve, 86
graph algorithms, 173
  see also breadth-first search, Dijkstra's algorithm
graph theory, 151, 172, 173
graphs, 151, 169
  see also die-hard graph, Hanoi graph, move graph, push graph, wolf-goat-cabbage graph
  representation, 152–153
  visualization, 151
gzip program, 273

H

Hamiltonian cycles, 156, 170, 412
Hamiltonian paths, 170, 412
Hanoi, 368
Hanoi graphs, 169–170
harmonic mean, 325
Harvey-van der Hoeven multiplication, 329
hash codes, 119, 253, 339
hash functions, 253–255, 279
hash tables, 152, 339, 454
hexadecimal numbers, 91, 276
higher-order functions, 159, 346
Hilbert curves, 65–70, 86
histograms, 36, 38, 360
Horner's scheme, 255, 309, 311
Huffman codes, 226, 260, 268, 269, 272, 273
  compared to other codes, 230, 231, 235, 236
  for byte sequences, 228–230
  for Lempel-Ziv tokens, 263–266
  limitations, 237
  overview, 223–225
Huffman trees, 225–227, 441, 443
  large, 260
  maximal height, 230
Huffman's algorithm, 225–227, 237, 431, 441
  deterministic implementation, 227

implementation, 226–228
worst case, 431
HuffmanTree, 225, 226, 431

I

identity matrix, 51
identity transformation, 51, 53, 55
IFS, *see* iterated function systems
imaginary part, 24
imaginary unit, 24, 25
immutable classes, 280, 283
importing classes, 334
importing static members, 335
induction, 170, 288, 291, 353, 369, 410
infinite sequences, 23, 25, 26
information theory, 235
integer overflow, 97, 285, 316–317, 380, 381
integer powers, 320–322, 381
integer types, 90, 96
integers modulo m, 327
interfaces, 340–342
interpolation, *see* linear interpolation
intersection
see also set operations
of axis-aligned rectangles, 16
iterated function systems, 7, 8, 21
see also affine iterated function systems
iteration, 1, 45–48, 62–63
iteration limit, 28, 31, 38, 360
iterators, 113–114, 344–345

J

Java, 333–352
jealous couples problem, 168–169
Julia sets, 39–40, 43

K

Karatsuba, 295
Karatsuba multiplication, 294–298, 320, 328
Kernighan's algorithm, 381
Koch snowflake, 78

L

lambda expressions, 115, 162, 347–349, 351
modifying local variables, 348
layers of a graph, 160, 196, 401
leading zeros, *see* number of leading zeros
least significant 1-bit, 112, 115
Lempel-Ziv compression, 238
Lempel-Ziv-Markov chain algorithm, 273
lerp function, 5, 7, 19, 21, 73, 75, 81
Life, *see* Game of Life
life forms, 40–41
line feed, 214
linear interpolation, 5, 19–21, 359
see also lerp function
linear transformations, 10
effect on area, 17
lists, reversing, 48
little-endian order, 277, 278
locality-preserving mapping, 86
logarithm, binary, *see* binary logarithm
longest shortest path, *see* diameter
lookahead region, 238, 248, 250
lossless compression, 212, 273
lossy compression, 273
LWSS life form, 41
LZ4 compression, 270–272
LZ77, 238, 273
bit encoding, 256–268
compression, 239–240, 243
decompression, 241–243
finding the longest match, 251–256
generating tokens, 248–250
token representation, 244
tokens, 239, 246
LZ77-Compress, 241, 243, 439
LZ77-Decompress, 243, 439
LZ78 compression, 273
LZH encoding, 263–268
LZMA compression, 273
LZSS encoding, 256–259, 268, 269
LZW compression, 273

M

Machin's formula, 325, 329, 330
Mandelbrot iteration, 26, 27, 30, 33, 35, 360
Mandelbrot set, 23–24, 26, 29, 35, 43
see also Buddhabrot, Julia sets
coloring, 26–29, 33
computing, 24–26
drawing, 30–33
landmarks, 29
Mandelbrot-BW, 26
Mandelbrot-Color, 30
master theorem, 320

memoization, 137, 140–144
memory partitioning, 104
method overriding, 339–341
method references, 347, 351
midpoint, 5, 374, 377
minimum in a list, 46
mixed-radix system, 323
modulo operator, 68, 247, 255, 362, 429
Morse code, 211, 212
move graph, 183–186, 188, 192, 196, 417
multigraphs, 155
multiplication
  algorithms compared, 328–329
  Karatsuba's algorithm, 294–298
  of big integers, 290–292
  standard algorithm, 288–292
MULTIPLY, 290, 291, 294, 295
MULTIPLYBYDIGIT, 289, 291, 446

N

Nebulabrot, 38, 361
Nim, 143–145, 147
Nim sum, 144, 398
Nimatron, 147
nlz function, *see* number of leading zeros
nodes in a graph, 151
normalization, 137, 138, 144, 404, 406
NOT operator, 110
NP-hard problems, 148, 162
ntz function, *see* number of trailing zeros
number of leading zeros, 103, 105, 106, 122, 222
number of trailing zeros, 103, 112, 113, 121

O

object-oriented design, 183
OR operator, 93, 107, 109
overflow, *see* integer overflow

P

P (complexity class), 209
packages in Java, 333
pandigital numbers, 452
parametric curves, 71, 79
  examples, 84
  plotting, 84–85
path finding, 185, 190–192, 202–204, 207, 208
  see also breadth-first search, Dijkstra's algorithm, A* algorithm
path in a graph, 152, 186, 190
  length, 152, 166
  longest shortest, 162
  representation, 164–166
  shortest, 164–167, 174, 175, 185, 192, 418
path in a weighted graph
  representation, 200
  shortest, 194, 197–202
  weighted length, 194
PBM file format, 94
Peano curve, 86
peg solitaire, 125, 142, 146
  computing moves, 129–132
  counting solutions, 137, 141
  difficulty of, 141, 142, 395
  representation of boards, 126–127
  solving, 134, 136
  symmetries, 137, 138
  variations, 132
Penrose tilings, 81–83, 87
period of decimal expansion, 322, 323
periodic boundary conditions, 41, 116, 117
pi, computing, 324–327, 329–330
plants, modeling of, 59
PNG file format, 273
points vs. vectors, 4
polar coordinates, 18, 37, 59
polydivisible numbers, 322
polynomial time, 209
population count, 115, 119–121, 451
position vectors, 4
positional number systems, 276
powers of 2, 105
  bit patterns, 99
  bit patterns of negative, 99
  computing with, 100–107
  dividing by, 101–102, 385
  divisibility, 102–103
  multiplying by, 100
  partitioning memory, 104
  testing for, 115
predecessor on a path, 164, 166, 198, 200
predicates, 157–160, 166, 167, 202, 351–352
prefix codes, 214–217, 219, 223, 231, 234
  see also Huffman codes
  bit encoding, 221–222
  converting representations, 218, 219
  decoding symbols, 215, 220

encoding symbols, 220
implementation, 217–222
optimal, 223
tree representation, 215, 216, 223
prefix of a string, 214, 215
prefix-free codes, *see* prefix codes
primitive types, 246, 335–337
priority queues, 198, 201, 227, 228, 344
data structures for implementing, 208
PSPACE-completeness, 209
pure function, 141
push graph, 186–189, 191, 194, 418
visualization, 187
Pythagoras trees, 48–51, 56–59
Pythagorean theorem, 49, 353

Q

queues, 156, 157, 197
quotient, 299

R

R-pentomino life form, 41
radius of a graph, 162, 163
radix, 276, 281, 283, 299, 303, 305
conversion, 308–313
for big integers, 276–277
multiplying by power of, 294, 297
radix conversion, 313, 448
random number generators, 6, 14, 37, 351
random numbers, 4, 13–15
random point inside a circle, 37
rasterization, 43
rational numbers, 322
real part, 24
records, 339–340
recursion, 45–48, 57–63, 78, 80, 133, 143, 368, 369, 400, 410
see also backtracking, divide-and-conquer method, memoization
eliminating, 122
implementation of, 62
infinite, 377, 447
optimization of, 63, 137, 142
relation to trees, 65
runtime analysis, 320
reflection of points, 353
regular polygons, 18
remainder, 101, 299
reversing bytes, 94
Rhind Papyrus, 329
river-crossing puzzles, 173
ROT13 cipher, 429
rotation, 10, 53
rotation matrix, 53
rule of a cellular automaton, 116
runtime complexity, 291, 295, 320
Russian peasant multiplication, 328

S

Sardinas-Patterson algorithm, 236
satisfiability problem, 148
scaling, 10, 53
scene graphs, 63
Schönhage-Strassen multiplication, 328, 329
self-similarity, 3, 67, 356
set operations, 109–112
setter methods, 338
seven bridges of Königsberg, 172
Shannon-Fano codes, 230–231
shearing, 10
Sierpiński triangle, 2, 3, 6, 8, 116, 354, 410
sign extension, 97, 101
sign-magnitude representation, 314–319, 454
simple graphs, 155
singleton sets, 107
snowflake, *see* Koch snowflake
Sokoban, 175–204
computational complexity, 209
deadlocks, 189–190
difficulty of, 176, 209
Level 1, 176
moving crates, 186–190
moving the player, 182–185
overview, 175–176
representing levels, 177–182
solving, 190–194, 202–204
text encoding of levels, 180–181
source coding theorem, 235
sources in a graph, 401
space-filling curves, 68, 86
spaceships in Life, 41, 362
spreadsheet, cell names, 313
squared length, 31
stack overflow, 63
subdivision surfaces, 86
SUBTRACT, 286–288, 291

subtraction, 286–288
Sudoku, 145, 148
suppliers, 350–351
symmetric difference, 381
systems programming, 122

T

tabulator, 214
tail-call optimization, 63
telegraphy, 211
tessellations, 81
three-way comparison, 281, 319, 352
tilings, 81, 87
TO-RADIX, 313, 450
Toom-Cook algorithm, 328
Towers of Hanoi, 61–63, 169, 170
trailing zeros, *see* number of trailing zeros
translation, 10, 51, 53
Turing completeness, 41
two's-complement representation, 95–100, 105, 112, 115, 287, 380, 381
  asymmetry, 96, 315
  changing bit width, 97
  common bit patterns, 97–99
  negation, 96
  overflow, 316–317
  trailing zeros, 103
type casts, 283, 336
type inference, 347
type promotion, 335

U

unambiguous codes, 217, 236
unary code, 217, 261, 268, 274
unboxing of primitive data types, 337
undirected edges, 151
Unicode, 232–234
union, *see* set operations
unit circle, 37
universal codes, 274
universe, age of the known, 369
UTF-8, 232–234

V

variable-length codes, 214, 233
  see also variable-length integers
variable-length integers, 261, 263, 268, 274, 441
vertices in a graph, *see* nodes in a graph
visibility of fields and methods, 335

W

water measuring problems, 169, 173
  see also die-hard problem
wavelet transforms, 273
Weierstrass function, 42
weighted average, 5
weighted graphs, 194, 197, 198
  see also Dijkstra's algorithm, paths in a weighted graph
weighted push graph, 194–196, 202
whirl patterns, 59
winning strategy, 143, 147
wolf-goat-cabbage problem, 168
word ladders, 204–205
word-ladder graph, 204
wrapper classes, 336–337

X

XOR operator, 112, 144, 147
xz program, 268, 273

Z

z-order curve, 86
ZIP file format, 273
zstandard compression, 273