Daniel Heck

PROGRAMMING IDEAS

Programming Ideas

Daniel Heck

Programming Ideas

http://programmingideas.net

First Edition

Revision 2 (July 15, 2026)

https://programmingideas.net

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.