Title: Foundations of Data Structures and Algorithms

Author(s): Patrick Kellogg

1. About the author

Patrick Kellogg has an undergraduate degree in electrical engineering from the University of Minnesota and a graduate degree in computer science from the University of Colorado at Boulder. He has worked for over ten years as a professional programmer for many Colorado companies, including Qwest, Corporate Express, Coors, and the National Center for Atmospheric Research. He has written code in C/C++, Java, Visual Basic, COBOL, Pascal, Fortran, perl, MATLAB, and assembler, and recently finished teaching a college class on software engineering. In addition, he written for two technical magazines and has won several English awards, including an essay scholarship from the National Council of Teachers of English and a science fiction short-story contest.

2. Summary

Foundations of Data Structures and Algorithms is a tutorial for programmers who may not have a formal background in software engineering, but still want to learn about modern algorithmic techniques. The book will cover, at a high level, most of the information presented in a data structures and algorithms class taught at a four-year university, in a clear fashion without a heavy reliance on mathematical proofs.

3. Description

Many college textbooks split data structures and algorithms into two separate sections. Then, they often divide those sections further. While this makes for an easy book to *write*, it forces the student to read many chapters about data structures before explaining how they could be used. Instead, we will introduce a new data structure like a “binary tree” in order to explain searching or sorting through data. Or, to name another example, an introduction to “queues” might be used to explain scheduling and then move on to greedy algorithms in general.

Foundations of Data Structures and Algorithms will be “language agnostic”, and won’t prefer one language over another. We will have code samples in four or five languages for each algorithm. Since algorithms are usually pretty short, adding the extra code will not take up a lot of page space. The code will be well-written for each language, using actual naming conventions and programming standards: avoiding variables named “X”, and starting all arrays at “0” and not “1”, to name some common pseudocode errors.

Pseudocode is often very useful in an academic setting, but not so useful when a reader “just wants something to work”. In a worst-case scenario, the reader might translate some pseudocode into a function for their program, only to find it doesn’t work. In that case, the reader might not know if the error lies in their translation, the pseudocode (a common problem, since the pseudocode can’t be run or tested in that form and often gets published with bugs), or in a fault with the entire algorithm itself. It would be better to have pre-tested code samples the user could cut-and-paste, avoiding possible errors. After a section on the theory (how the algorithm works), is an "implementation notes" section that introduces caveats that someone needs to know when implementing the algorithm in a particular language. For example, if the area under discussion is implementing a heap using an array, a potential Java implementation note would be that Java will throw an ArrayIndexOutOfBoundsException if you run off the end of the array.

This book is meant to be practical, so we will leave out almost all mathematical proofs (except for occasional “digressions” in the margins). After reading this book, the reader will have a passing knowledge of almost all of the “important” ideas in computer science as well as code samples to implement those data structures and algorithms. If a term such as “memoization” comes up at work, the programmer could use our book as a reference and explanation of the term, with examples and sample code.

4. Competing or comparable books

One of the leading algorithms textbooks, "Introduction to Algorithms" by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, is 1028 pages long, and retails for about $70. The first six chapters and 135 pages are used to set up the mathematics for the rest of the book. Also, the authors recommend that the book should be accompanied by a preliminary book on data structures and possibly a language reference. So, the reader must read the equivalent of two or three texts before they reach the start of the material on algorithms.

Instead, Foundations of Data Structures and Algorithms will not be a textbook that would need a professor’s help to explain. It will have a “plug and play” feel, similar to the “Numerical Recipes” series from Cambridge University. Readers should be able to find algorithms that are applicable for their work, and should be able to read a description and understand the basics of the algorithm easily. Finally, they should be able to find a code sample in the language of their choice to include in their program. The book should be instantly useful for solving programming problems, while educating the reader about the use of data structures and algorithms.

Probably the most famous series of algorithms books is Don Knuth’s “The Art of Programming”. While not merely about algorithms, the three volume set (with a fourth on the way) is the exhaustive reference for sorting algorithms and random number generation, among other things. However, all examples are written in Knuth’s arcane “MIX” assembly language, and its 2000+ page length would seem intimidating to any computer beginner. Likewise, Robert Sedgewick’s “Algorithms in C++” comes in a five-volume bundle, at 1200 pages. While Sedgewick’s new 3rd edition is a vast improvement over the first, there are still problems with consistency of tone and style. It simply feels like a book that has been written over the course of several years without taking into account new changes in computer languages over that time, with C++ stuck in where C code used to be.

There is a need for a smaller, single-volume overview of algorithms written for a user without a heavy mathematical content. There are many other books that deal with data structures and even more that deal solely with algorithms, but an approachable book that interleaves the two would be useful. To force the reader to wade through several dry introductory chapters discussing data structures is to remove the beauty and simplicity of a lot of the modern algorithms that we use today. We hope to distinguish our book by making it extremely readable, practical, fun, and “user friendly”.

5. Structure

Each chapter will open up with a problem that the new data structure and/or algorithm will solve. For example, having covered how to use dynamic programming as a way to figure out how to multiply a system of equalities, the next chapter will ask how to solve a system of inequalities. The chapter’s content on linear programming, will then provide a solution. So, the reader’s interest is peaked by asking questions that that they might find in their daily programming work.

Q: Why should I care about the speed of an algorithm?

A: An explanation of O-notation and economies of scale

Q: How do I sort things quickly?

A: An explanation of simple sorting algorithms leads to a description of the fastest possible modern sorting techniques (and hints of future methods such as quantum or DNA computing)

Q: Why not leave my data in an array and work with it there?

A: This provides a great opportunity to introduce a new data structure, such as a binary tree, and show how it can find minimum and maximum elements faster than an array search

Then, after introducing the question, the chapter will proceed to describe a new data structure or algorithm that will solve the problem. We will do this by describing the object clearly, and by providing plenty of examples, diagrams, and code samples. There are other "elements" that will be incorporated regularly through the book to make the book more browsable and varied. These will often show up in the margin of the book, if there is room, and will be highlighted in a way to distinguish them from the rest of the text.

> Anecdote

A brief, and usually amusing, digression that bears some relation to the topic at hand. It is separated out to avoid confusing the idea of the main text

> Puzzle

Used to exercise the reader's right brain, or provide the “aha” expression on a particular topic. These are very effective when combined with an illustration

> Case Study: "In the wild"

Shows how the topic is (or has been) used in the real world. Who cares about traveling salesmen? How do I route a packet across a network? Which problems are still outstanding and “NP complete”? Which algorithms would rarely be used, I favor of another approach?

> Exercise

In many senses, the opposite of a puzzle. Rather than trying to elicit insight, exercises drill the reader to make the information stick. There will be answers to the exercises in an appendix at the back of the book

> Code Sample

Provides illustrations of exactly how an algorithm or data structure is implemented. A blueprint for the reader to use in his or her own work. Code will be provided for four languages, with examples on top of each other, so the user can see at a glance how the algorithms might differ for each language. Occasionally, code will be followed by an “implementation note” that will elaborate on a trick or problem with a certain language (for example, perl has a built in “hash” type, so there is no need for the user to build their own)

> Terminology

Formal introduction of a term that is either recurrent or important (e.g., “Node”). These terms will also appear in the index at the back of the book

> Mathematical digression

Probably won't be used by most of the readers, but there will be readers who come from a math background (math, physics, chemistry, or maybe philosophy) and will appreciate a brief discussion of the mathematical merits of the book. Also, non-math readers may skip over it initially, but come back to it after a year or two. Separating out the math will help the main text be uncluttered, while still providing “meat” for the rest of the readers.

> Downfall

This is a “pitfall” or common error that a beginning programmer might not realize when initially using the data structure or algorithm. For example, an algorithm in C++ might use both “if a = b” to see if the element a is at the same memory location as element b, and the user should be careful not to accidentally type “if a = = b”

For this book to be maximally helpful, it will have to have lots of exercises with clear and detailed solutions. Algorithms and data structures are hard, and it's going to take some convincing to get the reader to go from chapter 2 to chapter 3 and to continue on. Putting in exercises gives the reader a metric with which to measure their progress, as well as providing measurable goals. Also, the reader feels more engaged with the material and develops a sense of pride from completing the exercises.

6. Readers

We will assume that the reader already knows how to program, and has a favorite language. Perhaps they have had one or semesters of a programming language and are programming as part of their job. Or, they could have a degree from a technical or community college. Other readers may be an expert in another field such as biology or astronomy, but have found themselves programming more and more without a background in formal computer science.

7. Book size and illustrations

480 pages over 12 chapters

Code examples for every algorithm and data structure

Code will be written with correct style and conventions for that language

Code will be implemented in four languages: C++, java, perl, and Visual Basic

There will be “tags” in each of the four language examples, so the text can refer to “section A” or “line B” of the code regardless of the language

Each data structure will be illustrated, such as the idea of “binary trees” or “queues”

As many algorithms as possible will be shown “while running” in graphical form

8. Tentative table of contents with annotations

FOUNDATIONS OF DATA STRUCTURES AND ALGORITHMS

(Patrick Kellogg) - TABLE OF CONTENTS

How to read this book

Multi-language code samples

Introduction. Algorithms

This chapter will answer the question, “What are algorithms and why should I care?” We will provide visceral reasons why new programmers should learn about proven and tested techniques to make their programs run faster. Using analogies, puzzles, and demonstrations of “economies of scale”, this chapter provides a foundation for the rest of the book to talk about what it means to analyze or compare the performance of different algorithms.

Q: What is an algorithm?

A: A way to do something

A: A *fast* way to do something

A: A *fast* and *pre-tested* was to do something (don’t need to reinvent the wheel)

Speed vs. clarity (digression on “design patterns”/ expected ways to do things)

Speed is important for scalability (bad algorithms may work at first, but not later)

How to compare speeds (constant time vs. linear vs. logarithmic vs. exponential)

Graphs on importance of orders of magnitude for algorithms

“Big O” or “O-notation”

Why O(3n) is really O(n)

Anecdotes: about importance of speed

Space requirements as well as speed

Downfall: how to ruin a perfectly good algorithm by accidentally increasing O(1) to O(n)

Chapter 1. Translating existing data into a data structure

Instead of jumping into data structures like most computer science textbooks, this chapter will talk a little about data itself, and then provide reasons why data could be placed in a data *structure* for better performance and clarity. This chapter will be very easy to understand, in order to smooth out the learning curve and to get the reader “up and running” with data structures. By the end of this chapter, the user will have actual code loaded into a framework they can build onto later. In addition, this chapter introduces “greedy algorithms”, one of the easiest classes of algorithms which will utilize the newly learned data structures for practical results.

Flat files

What are flat files?

Why is it not necessarily a good idea to leave data in a flat file?

Anecdote: looking at data in a new way can solve a problem (like Polya’s “How To Solve It” or Bentley’s “Programming Pearls”)

Queues, heaps, and stacks (LIFO vs. FIFO)

Implementation as an array

Why is this sometimes bad?

Implementation as a linked list

Code example for linked list

Pictures of linked list

More on linked lists

Doubly linked lists

Skip lists

Scheduling

Greedy algorithms

Making change

Creating Huffman codes using a greedy algorithm

Chapter 2. Sorting

Sorting is a popular topic for introductory computer science textbooks, because it provides an easily understood example of a common programming task. By describing various sorting algorithms, the reader will see why certain ones are “better” than other algorithms, and they will get a hint of what is happening “under the hood” of the operating system’s memory. In practice, most programmers will *never* be called upon to write a sorting algorithm. However, it is a kind of litmus test for beginning programmers, and woe to the ones who haven’t learned about basic terminology and algorithm types. So, this chapter lets a programmer say “Oh yes, I’ve heard of a bubble sort, but I think a quicksort would be more appropriate for our code”.

Everybody starts with sorting

What is a heap?

Compare it to stack space

Binomial and Fibonacci heaps

Heap sort of a previously-discussed data structure

Code examples

Pictures of algorithm in execution

O(n2)

Insertion sort of an array

Code examples

Pictures of algorithm in execution

O(n2)

Shell’s Method variation

Merge sort of an array

Code examples

Pictures of algorithm in execution

O(n lg n)

Average-case, best-case, and worst-case

A passing mention of Ω- and Θ-notation (omega- and theta-notation)

Many different sorting algorithms

Selection sort

Bubble sort

Shellsort

Code examples

Pictures of algorithms in execution

Q: Which to use?

A: Depends on your data (e.g. already half-sorted? Must be “in-place”?)

A: None of these… we can do it faster on average using recursion

Chapter 3. Quicksort, recursion, and divide-and-conquer

Here is the most popular sorting algorithm, now that the reader is prepared for it. Plus, the quicksort algorithm provides an introduction for a lot of other interesting computer science topics, such as recursion. Again, recursion is more important for programmers to learn about than to actually use in practice, but it leads to other tools such as divide-and-conquer algorithms. By placing quicksort in its own chapter, instead of mixed in as “just another sorting method”, the chapter should be more interesting to read, more like a practical tutorial than a mathematical treatise.

Quicksort on arrays

Code examples

“Divide-and-conquer”

Recursion trees

Infinite series

Best-case performance

Worst-case performance

Average-case performance

Head- and tail-recursion, and replacing tail-recursion with iteration

Base case and recursion

Why does my program run out of memory?

Saving return addresses and local variables to the stack

Divide-and-conquer

Another divide-and-conquer example

Arithmetic as divide-and-conquer

Other fun addition, subtraction, multiplication, and division examples

Chapter 4. Trees

Trees are an extremely important topic, since they are used in so many ways: to sort things, create fast lookup tables, or to explain geometric graphs and networks. We start with the easiest tree, a binary tree, and explain it exhaustively. Then, when the user understands that new data structure, we describe other trees as variants of the main idea, rather than new data structures as many texts do. There’s a lot of material and terminology to cover in this chapter, which is why the material ends with a practical application of the idea: the use of trees to sort through large amounts of data.

Binary tree

Code examples for setting up a binary tree

Code examples for finding elements

Code examples for inserting elements

Code examples for deleting elements

In-depth code: binary tree as a heap or array

Terminology: nodes, roots, edges, subtrees, ancestors, descendents, parents, length, depth, and children

Downfall: unbalanced trees

Skip lists

“A forest of trees”

AVL Trees

Multiway trees

B-Trees

Splay trees

Red-black trees

2-3-4-Trees

Top-down 2-3-4-Trees

Code examples

Pictures of trees

Searching a tree

When you would use it (i.e. finding, inserting, and deleting elements)

Inorder, postorder, and preorder tree walks

Using symbol tables

Chapter 5. Searching, and artificial intelligence

The idea of searching through a tree leads naturally to the field of artificial intelligence. Many beginning computer science students are disappointed when they find out that the field of A.I. is merely a set of useful algorithms, instead of some sort of magic that makes computers automatically think like people. Some artificially intelligent applications would be find the “ideal” move in a game of checkers or chess, or to find a robot’s path through a crowded room. This leads to the idea of “agents” that have incomplete knowledge, which is the start of talking about neural networks and genetic algorithms. This is a very useful chapter, since it provides many different types of algorithms, and should provide a clear basis if the reader wants to look elsewhere for more detailed books.

Artificial intelligence is actually as set of algorithms to use (not magic)

The A\* algorithm

Greedy (depth-first) searching

Uniform-cost (breadth-first) searching

Iterative deepening search

Other strategies, such as uniform-cost, depth-limited, and bidirectional search

Heuristic (greedy + uniform) searching

Passing mention of IDA\* and SMA\*

Two-player games

Othello, not chess as an example

Minimax

Alpha-beta pruning

Passing mention of expectiminimax

Agents

Value and policy iteration

Reinforcement learning in general (like TD-lambda and Q-learning)

Iterative improvement

Hill climbing algorithms

Passing mention of neural networks and genetic algorithms

Chapter 6. Probability, approximation algorithms, and statistics

The material on neural networks and genetic algorithms in the preceding chapter might be some of the hardest math yet described in this book. So, the next chapter goes a little slower, describing how artificial intelligent algorithms can be improved by adding randomness, or by approximating the answer to give a “best guess”. Hopefully, this material will be clear and free from long equations, even though it is dealing with some tricky ideas. By the end of the chapter, the user should be ready for some real-world topics from numerical analysis, and will understand the code examples and be able to integrate them into their own work.

Randomized quicksort

Simulated annealing

Selection in linear time

Sorting in linear time

Counting sort, radix sort, and bucket sort

Discussion of NP-complete problems and real-time systems that need a “best guess”

Approximation algorithms

Runs in polynomial time and has a guaranteed “performance bound”

Turning NP into “pretty good”

Constant-factor solution: Vertex cover (or Steiner tree problem)

Graphs for vertex cover (or Steiner tree)

Code example for vertex cover (or Steiner tree)

Logarithmic-factor solutions: Survivable network design

Graphs for survivable network design

Code example for survivable network design

Polynomial-time solutions: Knapsack problem

Graphs for knapsack problem

Code example for knapsack problem

Statistical algorithms

Root finding with Newton’s method

Other numerical analysis algorithms: interpolation, numerical integration, others

Preprocessing data and visualization (like Edward Tufte’s excellent books)

Chapter 7. Digital signal processing and parallel algorithms

Again, this chapter is heavily mathematical, but that doesn’t mean it needs to be obtuse. Fourier transforms are a very useful tool for analyzing pictures, sounds, and scientific data. By converting information into the “frequency domain” the user may be able to answer questions that were difficult to visualize before. The FFT can be tricky, however, so a clear explanation is needed that will digress into how the algorithm can be handled by more than one computer in a “divide-and-conquer” format from the third chapter. This brings up all sorts of topics, from parallel processors inside a single computer, to distributed and “peer-to-peer” programs that run on many computers at the same time.

The Fourier Transform

Discussion of the time vs. frequency domains

The Fast Fourier Transform as a divide-and-conquer problem

The Discrete Fourier Transform

Some fun DSP algorithms

Pitch perception and pitch tracking (and zero-crossings)

Audio synthesis (analog, digital/FM, and sampling)

Examples from Netlib (http://www.netlib.org/) and other web resources such as

Harmony Central (http://www.harmony-central.com/)

Parallel algorithms

Great at solving divide-and-conquer

Sort in O(n) or even O(sqrt(n)) or O(log n) for mergesort

Comparison to previously discussed algorithms for searching and sorting

Batcher's sorting networks

Chapter 8. Dynamic programming

This is a difficult topic for new computer science students, so after they learn enough to pass a midterm exam, most of them will forget it and rarely use it in their professional lives. That’s a shame, since this tool is often the best way to solve many tough programming challenges. So, this chapter provides many examples of how dynamic programming could be used in a professional setting, perhaps with more examples than in previous chapters. The idea is to convince students that there is a class of problems that is ideally suited to dynamic programming, and hopefully they will be able to recognize those problems even years after reading this chapter.

Example of a problem that is best solved by dynamic programming

Memoization

Example of memorization like fast exponentiation

Binomial and Fibonacci heaps

Cost functions (like a heuristic?)

Constructing a table

Defining the recurrence

1-D algorithms

“Concave-cost” example like paragraphing in O(n) time

Or “economic lot-size” model problem

Assembly-line scheduling

Optimal binary search trees

2-D algorithms

Longest common subsequences (and spelling checkers)

Inventory management, production schedules, and resource allocation

Matrix multiplication

Amortized analysis

Hash tables

Open addressing

Chaining

Extendible hashing

Chapter 9. Graph algorithms and computation geometry

Graphs are very useful because they can be described without using mathematical equations, and because they are useful for modern networks and internet problems. Using the basics the reader has learned about tree data structures, this chapter expands the idea to talk about computer graphics and computational geometry. This chapter will be a potpourri of useful geometrical algorithms, and many pictures will help the reader understand how the code works and why they would want to use it.

Graphs

DAGs (directed acyclical graphs)

Weighted graphs

Dijkstra's algorithm

Matroids

Minimum spanning trees and connectivity of a network (bridges and cutpoints)

Prim and Kruskal’s algorithms

Gabow's strong component algorithm

Tarjan’s algorithm

(other related graph algorithm)

Networks

Real-world examples of graph theory in action

Distributed computing (like Napster)

Peer-to-peer computing

Computer graphics

Graham’s convex hull algorithm

3D Incremental Algorithm

Voronoi/ Delaunay triangulation (duals)

Closest-pairs as divide-and-conquer and sweep-lines

Clipping and segment-segment intersection algorithms

Mention of more complex intersections, like those of multiple polygons

Chapter 10. Mathematics and cryptography

Again, we will try to explain some important mathematical algorithms without resorting to columns and columns of formulae. The explanation of several useful algorithms will give the reader the ability to perform new functions on large sets of data. This exploration of “number theory” leads to some of the most practical uses of that field today: encryption and cryptography. Finally, this chapter talks more about the ideas of data and information that were brought up in the introduction of the book.

Math algorithms

Greatest common divisor

Chinese remainder theorem

Primes

Primality testing with Miller-Rabin

RSA public-key encryption

Diffie-Hellman Key Exchange

Integer factorization

Solving matrices

Row reduction, Cramer’s rule, and the inverse matrix method

Gaussian elimination, triangular matrices, and LU factorization

Partial pivoting (with code samples)

Dense and sparse matrix operations

Quaternions vs. Matrices

Cryptography

Number theory

Pseudo-random numbers

One-way hash functions

Authentication and key exchange

No need to write your own algorithm: i.e. DES (Data Encryption Standard)

Mention of Diffie-Hellman

Quantum and DNA computers

Information theory

What is random? Randomization tests

Complexity and chaos

Chapter 11. Linear Programming

Linear programming is turning out to be a very useful and important modern programming technique. However, it can also be hard to understand. So, we explain it by using several different examples that attack the problem from several angles: as a matrix algorithm, a geometric problem, and a memory issue. Tangible problems are solved in the fields of manufacturing, science, and accounting, that could only be solved easily by using this new method.

Description of linear programming (LP) as geometric problem

LP vs. ILP (integer linear programming)

Knapsack problem as an example

Full description of the Simplex algorithm

The standard form

Dictionaries, pivots, and variables

Passing mention of the ellipsoid method and the interior point algorithm

Three-dimensional simplex

Downfall: cycling

Implementation issues

Linear programming with tableaus and matrices

Applications

Separation of polyhedra

“Cutting stock” and relation to the knapsack problem

Chapter 12. Bioinformatics

Genetic engineering is a hot topic, and has many unique algorithms that would be easy and fun to explain. Even better, the code discussed here uses almost all of the ideas presented in the previous chapters. So, this chapter is a good summation of what the reader has learned as well as a good selling point and reason for buying this book.

Genome analysis as pattern matching

Pairs of sequences

Multiple sequences

Prediction

Statistics and random number generation

Limiting the model space

Using databases

“Read-ahead” and storage issues

Virtual memory and disk thrashing

Sequence alignment and Monte Carlo methods

Randomized algorithms for bioinformatics

Further reading and Internet sources

APPENDICES

Answers to exercises

Bibliography

Lookup table of what algorithm to use for a particular problem

Websites

Downloads

Index

2/21/2002

9. Contact information

Patrick Kellogg

3186 West 36th Avenue

Denver, CO 80211

Home: 303-480-0890

Email: kellogg@dimensional.com

Home Page: http://www.patrickkellogg.com