Algorithm Design Techniques

Recursion

Backtracking

Greedy

Divide and Conquer

Dynamic Programming

By
Narasimha Karumanchi

Designed by *Narasimha Karumanchi*

Acknowledgements

Mother and *Father*! It is impossible to thank you adequately for everything you have done, from loving me unconditionally to raising me in a stable household, where your persistent efforts and traditional values taught your children to celebrate and embrace life. I could not have asked for better parents or role-models. You showed me that anything is possible with faith, hard work and determination.

This book would not have been possible without the help of many people. I would like to express my gratitude to all the people who had provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and designing. In particular, I would like to thank the following individuals:

- *Mohan Mullapudi*, IIT Bombay, Architect, dataRPM Pvt. Ltd.
- *Navin Kumar Jaiswal*, Senior Consultant, Juniper Networks Inc.
- *Prof. Anuradha*, SriChaitanya Academy, Hyderabad.

–Narasimha Karumanchi
M-Tech, *IIT Bombay*
Founder, *CareerMonk.com*, *CodingDucks.org*

Preface

Dear Reader,

Please hold on! I know many people typically, who do not read the *preface* of a book. But I strongly recommend that you read this particular Preface.

The study of algorithms and data structures is central to understanding what computer science is all about. Learning computer science is not unlike learning any other type of difficult subject matter. The only way to be successful is through deliberate and incremental exposure to the fundamental ideas. A novice computer scientist needs practice and thorough understanding before continuing on to the more complexities of the curriculum. In addition, a beginner needs to be given the opportunity to be successful and gain confidence. This textbook is designed to serve as a text for a first course on data structures and algorithms. We looked at a number of data structures and solved classic problems that arose. The tools and techniques that you learn here will be applied over and over as you continue your study of computer science.

In all the chapters, you will see more emphasis on problems and analysis rather than on theory. In each chapter, you will first see the basic required theory followed by various problems. I have followed a pattern of improving the problem solutions with different complexities (for each problem, you will find multiple solutions with different, and reduced, complexities). Basically, it's an enumeration of possible solutions. With this approach, even if you get a new question, it will show you a way to *think* about the possible solutions. You will find this book useful for interview preparation, competitive exams preparation, and campus interview preparations.

For many problems, *multiple* solutions are provided with different levels of complexity. We started with the *brute force* solution and slowly moved towards the *best solution* possible for that problem. For each problem, we endeavor to understand how much time the algorithm takes and how much memory the algorithm uses.

It is recommended that the reader does at least one *complete* reading of this book to gain a full understanding of all the topics that are covered. Then, in subsequent readings you can skip directly to any chapter to refer to a specific topic. Even though many readings have been done for the purpose of correcting errors, there could still be some minor typos in the book. If any are found, they will be updated at *www.CareerMonk.com*. You can monitor this site for any corrections and also for new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

I wish you all the best and I am confident that you will find this book useful.

Source code: https://github.com/careermonk/algorithm-design-techniques.git

–Narasimha Karumanchi
M-Tech, *IIT Bombay*
Founder, *CareerMonk.com*, *CodingDucks.or*

Table of Contents

Organization of Chapters

What is this book about?

This book is about the fundamentals of data structures and algorithms – the basic elements from which large and complex software projects are built. To develop a good understanding of a data structure requires three things: first, you must learn how the information is arranged in the memory of the computer; second, you must become familiar with the algorithms for manipulating the information contained in the data structure; and third, you must understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application, you are able to make an appropriate decision.

The algorithms and data structures in this book are presented in the Python programming language. A unique feature of this book, when compared to the books available on the subject, is that it offers a balance of theory, practical concepts, problem solving, and interview questions.

Concepts + Problems + Interview Questions

The book deals with some of the most important and challenging areas of programming and computer science in a highly readable manner. It covers both algorithmic theory and programming practice, demonstrating how theory is reflected in real Python programs. Well-known algorithms and data structures that are built into the Python language are explained, and the user is shown how to implement and evaluate others.

The book offers a large number of questions, with detailed answers, so that you can practice and assess your knowledge before you take the exam or are interviewed.

Salient features of the book are:

- Basic principles of algorithm design
- How to represent well-known data structures in Python
- How to implement well-known algorithms in Python
- How to transform new problems into well-known algorithmic problems with efficient solutions
- How to analyze algorithms and Python programs using both mathematical tools and basic experiments and benchmarks

- How to understand several classical algorithms and data structures in depth, and be able to implement these efficiently in Python

Note that this book does not cover numerical or number-theoretical algorithms, parallel algorithms or multi-core programming.

Should I buy this book?

The book is intended for Python programmers who need to learn about algorithmic problem-solving or who need a refresher. However, others will also find it useful, including data and computational scientists employed to do big data analytic analysis; game programmers and financial analysts/engineers; and students of computer science or programming-related subjects such as bioinformatics.

Although this book is more precise and analytical than many other data structure and algorithm books, it rarely uses mathematical concepts that are more advanced than those taught at high school. I have made an effort to avoid using any advanced calculus, probability, or stochastic process concepts. The book is therefore appropriate for undergraduate students preparing for interviews.

Organization of chapters

Data structures and algorithms are important aspects of computer science as they form the fundamental building blocks of developing logical solutions to problems, as well as creating efficient programs that perform tasks optimally. This book covers the topics required for a thorough understanding of the concepts such as Recursion, Backtracking, Greedy, Divide and Conquer, and Dynamic Programming.

The chapters are arranged as follows:

1. ***Introduction***: This chapter provides an overview of algorithms and their place in modern computing systems. It considers the general motivations for algorithmic analysis and the various approaches to studying the performance characteristics of algorithms.

2. ***Algorithms Design Techniques***: In the previous chapters, we have seen many algorithms for solving different kinds of problems. Before solving a new problem, the general tendency is to look for the similarity of the current problem to other problems for which we have solutions. This helps us to get the solution easily. In this chapter, we see different ways of classifying the algorithms, and in subsequent chapters we will focus on a few of them (e.g., Greedy, Divide and Conquer, and Dynamic Programming).

3. ***Recursion and Backtracking***: *Recursion* is a programming technique that allows the programmer to express operations in terms of themselves. In other words, it is the process of defining a function or calculating a number by the repeated application of an algorithm.

   For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path (for example problems in the Trees and Graphs domain). If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different

path. Algorithms that use this approach are called *backtracking* algorithms, and backtracking is a form of recursion. Also, some problems can be solved by combining recursion with backtracking.

4. ***Greedy Algorithms***: A greedy algorithm is also called a *single-minded* algorithm. A greedy algorithm is a process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. The idea behind a greedy algorithm is to perform a single procedure in the recipe over and over again until it can't be done any more, and see what kind of results it will produce. It may not completely solve the problem, or, if it produces a solution, it may not be the very best one, but it is one way of approaching the problem and sometimes yields very good (or even the best possible) results. Examples of greedy algorithms include selection sort, Prim's algorithms, Kruskal's algorithms, Dijkstra algorithm, Huffman coding algorithm, etc.

5. ***Divide And Conquer***: These algorithms work based on the principles described below.
    a. *Divide* - break the problem into several subproblems that are similar to the original problem but smaller in size
    b. *Conquer* - solve the subproblems recursively.
    c. *Base case*: If the subproblem size is small enough (i.e., the base case has been reached) then solve the subproblem directly without more recursion.
    d. *Combine* - the solutions to create a solution for the original problem

    Examples of divide and conquer algorithms include Binary Search, Merge Sort, etc.…

6. ***Dynamic Programming***: In this chapter, we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, Divide & Conquer and Greedy methods). Dynamic Programming (DP) is a simple technique but it can be difficult to master. One easy way to identify and solve DP problems is by solving as many problems as possible. The term Programming is not related to coding; it is from literature, and it means filling tables (similar to Linear Programming).

APPENDIX Complexity Classes: In previous chapters, we solved problems of different complexities. Some algorithms have lower rates of growth while others have higher rates of growth. The problems with lower rates of growth are called easy problems (or easy solved problems) and the problems with higher rates of growth are called hard problems (or hard solved problems). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem. There are lots of problems for which we do not know the solutions.

In computer science, in order to understand the problems for which there are no solutions, the problems are divided into classes, and we call them *complexity classes*. In complexity theory, a complexity class is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem. The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes). This chapter classifies the problems into different types based on their complexity class.

In this chapter, we discuss a few tips and tricks with a focus on bitwise operators. Also, it covers a few other uncovered and general problems.

At the end of each chapter, a set of problems/questions is provided for you to improve/check your understanding of the concepts. The examples in this book are kept

simple for easy understanding. The objective is to enhance the explanation of each concept with examples for a better understanding.

Some prerequisites

This book is intended for two groups of people: Python programmers who want to beef up their algorithmics, and students taking algorithm courses who want a supplement to their algorithms textbook. Even if you belong to the latter group, I'm assuming that you have a familiarity with programming in general and with Python in particular. If you don't, the Python web site also has a lot of useful material. Python is a really easy language to learn. There is some math in the pages ahead, but you don't have to be a math prodigy to follow the text. We'll be dealing with some simple sums and nifty concepts such as polynomials, exponentials, and logarithms, but I'll explain it all as we go along.