Data Structures And Algorithmic Thinking With Python

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 of the people who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. 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.
- *A.Vamshi Krishna*, IIT Kanpur, Mentor Graphics Inc.
- *Cathy Reed, BA, MA,* Copy Editor
- *Kondrakunta Murali Krishna,* B-Tech., Technical Lead, HCL
- *Prof.Girish P.Saraph, Founder,Vegayan Systems,IIT Bombay*
- *Kishore Kumar Jinka,* IIT Bombay
- *Prof.Hsin – mu Tsai, National Taiwan University, Taiwan*
- *Prof.Chintapalli Sobhan Babu. IIT,Hyderabad*
- *Prof.Meda Sreenivasa Rao, JNTU, Hyderabad*

Last but not least, I would like to thank the *Directors* of *Guntur Vikas College*, *Prof. Y.V.Gopala Krishna Murthy* & *Prof. Ayub Khan* [*ACE Engineering Academy*], *T.R.C.Bose* [*Ex. Director* of *APTransco*], *Ch.Venkateswara Rao VNR Vignanajyothi* [*Engineering College, Hyderabad*], *Ch.Venkata Narasaiah* [*IPS*], *Yarapathineni Lakshmaiah* [*Manchikallu, Gurazala*], & *all our well – wishers* for helping me and my family during our studies.

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

Preface

Dear Reader,

Please hold on! I know many people typically 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 beginning computer scientist needs practice so that there is a thorough understanding before continuing on to the more complex parts 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. In this book, we cover abstract data types and data structures, writing algorithms, and solving problems. We look at a number of data structures and solve classic problems that arise. The tools and techniques that you learn here will be applied over and over as you continue your study of computer science.

It is not the main objective of this book to present you with the theorems and proofs on *data structures* and *algorithms*. 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.

As a *job seeker,* if you read the complete book, I am sure you will be able to challenge the interviewers. If you read it as an *instructor*, it will help you to deliver lectures with an approach that is easy to follow, and as a result your students will appreciate the fact that they have opted for Computer Science / Information Technology as their degree.

This book is also useful for *Engineering degree students* and *Masters degree students* during their academic preparations. In all the chapters you will see that there is more emphasis on problems and their analysis rather than on theory. In each chapter, you will first read about the basic required theory, which is then followed by a section on problem sets. In total, there are approximately 700 algorithmic problems, all with solutions.

If you read the book as a *student* preparing for competitive exams for Computer Science / Information Technology, the content covers *all the required topics* in full detail. While writing this book, my main focus was to help students who are preparing for these exams.

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.

For many problems, *multiple* solutions are provided with different levels of complexity. We start with the *brute force* solution and slowly move toward 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.

—Narasimha Karumanchi
M-Tech, *IIT Bombay*
Founder, *CareerMonk.com*

Other Books by Narasimha Karumanchi

- Data Structures and Algorithms Made Easy
- IT Interview Questions
- Data Structures and Algorithms for GATE
- Data Structures and Algorithms Made Easy in Java
- Coding Interview Questions
- Peeling Design Patterns
- Elements of Computer Networking

Table of Contents

CHAPTER 0

Organization of Chapters

0.1 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 available books 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 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.

0.2 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 in 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.

0.3 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 subjects such concepts as Linked Lists, Stacks, Queues, Trees, Priority Queues, Searching, Sorting, Hashing, Algorithm Design Techniques, Greedy, Divide and Conquer, Dynamic Programming and Symbol Tables.

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. ***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.
3. ***Linked Lists***: A *linked list* is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list. It is a very common data structure that is used to create other data structures like trees, graphs, hashing. etc.
4. ***Stacks***: A *stack* abstract type is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle. There are many applications of stacks, including:
   a. Space for function parameters and local variables is created internally using a stack.
   b. Compiler's syntax check for matching braces is implemented by using stack.
   c. Support for recursion.
   d. It can act as an auxiliary data structure for other abstract data types.
5. ***Queues***: *Queue* is also an abstract data structure or a linear data structure, in which the first element is inserted from one end called as *rear* (also called *tail*), and the deletion of the existing element takes place from the other end, called as *front* (also called *head*). This makes queue as FIFO data structure, which means that element inserted first will also be removed first. There are many applications of stacks, including:
   a. In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes.
   b. Computer systems must often provide a *holding area* for messages between two processes, two programs, or even two systems. This holding area is usually called a *buffer* and is often implemented as a queue.
   c. It can act as an auxiliary data structure for other abstract data types.
6. ***Trees***: A *tree* is an abstract data structure used to organize the data in a tree format so as to make the data insertion or deletion or search faster. Trees are one of the most useful data structures in computer science. Some of the common applications of trees are:
   a. The library database in a library, a student database in a school or college, an employee database in a company, a patient database in a hospital, or basically any database would be implemented using trees.
   b. The file system in your computer, i.e. folders and all files, would be stored as a tree.
   c. And a tree can act as an auxiliary data structure for other abstract data types.

A tree is an example of a non-linear data structure. There are many variants in trees, classified by the number of children and the way of interconnecting them. This chapter focuses on some of these variants, including Generic Trees, Binary Trees, Binary Search Trees, Balanced Binary Trees, etc.

7. ***Priority Queues***: The *priority queue* abstract data type is designed for systems that maintain a collection of prioritized elements, where elements are removed from the collection in order of their priority. Priority queues turn up in various applications, for example, processing jobs, where we process each job based on how urgent it is. For example, operating systems often use a priority queue for the ready queue of processes to run on the CPU.
8. ***Graph Algorithms***: Graphs are a fundamental data structure in the world of programming. A graph abstract data type is a collection of nodes called *vertices*, and the connections between them called *edges*. Graphs are an example of a non-linear data structure. This chapter focuses on representations of graphs (adjacency list and matrix representations), shortest path algorithms, etc. Graphs can be used to model many types of relations and processes in physical, biological, social and information systems, and many practical problems can be represented by graphs.
9. ***Disjoint Set ADT***: A disjoint set abstract data type represents a collection of sets that are disjoint: that is, no item is found in more than one set. The collection of disjoint sets is called a partition, because the items are partitioned among the sets. As an example, suppose the items in our universe are companies that still exist today or were acquired by other corporations. Our sets are companies that still exist under their own name. For instance, "*Motorola*," "*YouTube*," and "*Android*" are all members of the "*Google*" set.

   This chapter is limited to two operations. The first is called a *union* operation, in which we merge two sets into one. The second is called a *find* query, in which we ask a question like, "What corporation does Android belong to today?" More generally, a *find* query takes an item and tells us which set it is in. Data structures designed to support these operations are called *union/find* data structures. Applications of *union/find* data structures include maze generation and Kruskal's algorithm for computing the minimum spanning tree of a graph.
10. ***Sorting Algorithms***: *Sorting* is an algorithm that arranges the elements of a list in a certain order [either ascending or descending]. The output is a permutation or reordering of the input, and sorting is one of the important categories of algorithms in computer science. Sometimes sorting significantly reduces the complexity of the problem, and we can use sorting as a technique to reduce search complexity. Much research has gone into this category of algorithms because of its importance. These algorithms are used in many computer algorithms, for example, searching elements and database algorithms. In this chapter, we examine both comparison-based sorting algorithms and linear sorting algorithms.
11. ***Searching Algorithms***: In computer science, *searching* is the process of finding an item with specified properties from a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or elements of other search spaces.

    Searching is one of the core computer science algorithms. We know that today's computers store a lot of information, and to retrieve this information we need highly efficient searching algorithms. There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in proper order, it is easy to search the required element. Sorting is one of the techniques for making the elements ordered. In this chapter we will see different searching algorithms.
12. ***Selection Algorithms***: A *selection algorithm* is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistic). This includes finding the minimum, maximum, and median elements. For finding k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities. We will also look at a linear algorithm for finding the k^{th} element in a given list.
13. ***Symbol Tables*** (*Dictionaries*): Since childhood, we all have used a dictionary, and many of us have a word processor (say, Microsoft Word), which comes with a spell checker. The spell checker is also a dictionary but limited in scope. There are many real time examples for dictionaries and a few of them are:
    a. Spelling checker
    b. The data dictionary found in database management applications
    c. Symbol tables generated by loaders, assemblers, and compilers
    d. Routing tables in networking components (DNS lookup)

In computer science, we generally use the term 'symbol' table rather than dictionary, when referring to the abstract data type (ADT).

14. ***Hashing***: *Hashing* is a technique used for storing and retrieving information as fast as possible. It is used to perform optimal search and is useful in implementing symbol tables. From the *Trees* chapter we understand that balanced binary search trees support operations such as insert, delete and search in O(*logn*) time. In applications, if we need these operations in O(1), then *hashing* provides a way. Remember that the worst case complexity of hashing is still O(*n*), but it gives O(1) on the average. In this chapter, we will take a detailed look at the hashing process and problems which can be solved with this technique.
15. ***String Algorithms***: To understand the importance of string algorithms, let us consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome). You will observe that after typing the prefix of the URL, a list of all possible URLs is displayed. That means, the browsers are doing some internal processing and giving us the list of matching URLs. This technique is sometimes called *auto-completion*. Similarly, consider the case of entering the directory name in a command line interface (in both Windows and UNIX). After typing the prefix of the directory name, if we press tab button, we then get a list of all matched directory names available. This is another example of auto completion.

    In order to support these kinds of operations, we need a data structure which stores the string data efficiently. In this chapter, we will look at the data structures that are useful for implementing string algorithms. We start our discussion with the basic problem of strings: given a string, how do we search a substring (pattern)? This is called *string matching problem*. After discussing various string matching algorithms, we will see different data structures for storing strings.
16. ***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).
17. ***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.
18. ***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….
19. ***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).
20. ***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 solutions are not there, 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.

21. ***Miscellaneous Concepts***: ***Bit − wise Hacking***: The commonality or applicability depends on the problem in hand. Some real-life projects do benefit from bit-wise operations.

    Some examples:

    - You're setting individual pixels on the screen by directly manipulating the video memory, in which every pixel's color is represented by 1 or 4 bits. So, in every byte you can have packed 8 or 2 pixels and you need to separate them. Basically, your hardware dictates the use of bit-wise operations.
    - You're dealing with some kind of file format (e.g. GIF) or network protocol that uses individual bits or groups of bits to represent pieces of information.
    - Your data dictates the use of bit-wise operations. You need to compute some kind of checksum (possibly, parity or CRC) or hash value, and some of the most applicable algorithms do this by manipulating with bits.

    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.

0.4 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 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.