Data Structure And Algorithmic Thinking With Python

By
Narasimha Karumanchi

Copyright© 2015 by *CareerMonk.com*

All rights reserved.

Designed by *Narasimha Karumanchi*

Copyright© 2015 CareerMonk Publications. All rights reserved.

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without written permission from the publisher or author

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 you persistent efforts traditional values and 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 many people who saw me through this book, to all those 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.
- *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 *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 do not read the preface. But I would strongly recommend that you go through the preface of this book at least. The reason for this is that this preface has *something different* to offer.

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.

The main objective of the book is not to give you the theorems and proofs about *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 all possible solutions for a given problem. This book is very useful for interview preparation, competitive exams preparation, and campus interview preparations.

In all the chapters you will see more importance given to problems and analyzing them instead of concentrating more on theory. For each chapter, first you will see the basic required theory and then followed by problems.

For many problems, *multiple* solutions are provided with different levels of complexities. We start with *brute force* solution and slowly move towards the *best solution* possible for that problem. For each problem we will try to understand how much time the algorithm is taking and how much memory the algorithm is taking.

It is *recommended* that the reader does at least one complete reading of this book to get full understanding of all the topics. In the subsequent readings, you can go directly to any chapter and refer. Even though, enough readings were given for correcting the errors, there could be some minor typos in the book. If any such typos are found, they will be updated at *CareerMonk.com*. I request you to constantly monitor this site for any corrections, new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

Wish you all the best. I am sure that you will find this book useful.

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

Other Titles 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

0. Organization of Chapters ---- 13
0.1 What Is this Book About? ---- 13
0.2 Should I Take this Book? ---- 13
0.3 Organization of Chapters ---- 14
0.4 Some Prerequisites ---- 17
1. Introduction ---- 18
1.1 Variables ---- 18
1.2 Data types ---- 18
1.3 Data Structures ---- 19
1.4 Abstract Data Types (ADTs) ---- 19
1.5 What is an Algorithm? ---- 19
1.6 Why Analysis of Algorithms? ---- 20
1.7 Goal of Analysis of Algorithms ---- 20
1.8 What is Running Time Analysis? ---- 20
1.9 How to Compare Algorithms? ---- 20
1.10 What is Rate of Growth? ---- 20
1.11 Commonly used Rate of Growths ---- 21
1.12 Types of Analysis ---- 22
1.13 Asymptotic Notation ---- 22
1.14 Big-O Notation ---- 22
1.15 Omega-Ω Notation ---- 24
1.16 Theta-Θ Notation ---- 24
1.17 Why is it called Asymptotic Analysis? ---- 25
1.18 Guidelines for Asymptotic Analysis ---- 25
1.19 Properties of Notations ---- 27
1.20 Commonly used Logarithms and Summations ---- 27
1.21 Master Theorem for Divide and Conquer ---- 27
1.22 Problems on Divide and Conquer Master Theorem ---- 28
1.23 Master Theorem for Subtract and Conquer Recurrences ---- 29
1.24 Variant of subtraction and conquer master theorem ---- 29
1.25 Method of Guessing and Confirm ---- 29
1.26 Amortized Analysis ---- 30
1.27 Problems on Algorithms Analysis ---- 31
2. Recursion and Backtracking ---- 42
2.1 Introduction ---- 42
2.2 What is Recursion? ---- 42
2.3 Why Recursion? ---- 42
2.4 Format of a Recursive Function ---- 42
2.5 Recursion and Memory (Visualization) ---- 43

2.6 Recursion versus Iteration ---- 43
2.7 Notes on Recursion ---- 44
2.8 Example Algorithms of Recursion ---- 44
2.9 Problems on Recursion ---- 44
2.10 What is Backtracking? ---- 45
2.11 Example Algorithms of Backtracking ---- 45
2.12 Problems on Backtracking ---- 45
3. Linked Lists ---- 48
3.1 What is a Linked List? ---- 48
3.2 Linked Lists ADT ---- 48
3.3 Why Linked Lists? ---- 48
3.4 Arrays Overview ---- 48
3.5 Comparison of Linked Lists with Arrays and Dynamic Arrays ---- 50
3.6 Singly Linked Lists ---- 50
3.7 Doubly Linked Lists ---- 56
3.8 Circular Linked Lists ---- 61
3.9 A Memory-Efficient Doubly Linked List ---- 67
3.10 Unrolled Linked Lists ---- 68
3.11 Skip Lists ---- 72
3.12 Problems on Linked Lists ---- 75
4. Stacks ---- 96
4.1 What is a Stack? ---- 96
4.2 How Stacks are used? ---- 96
4.3 Stack ADT ---- 97
4.4 Applications ---- 97
4.5 Implementation ---- 97
4.6 Comparison of Implementations ---- 101
4.7 Problems on Stacks ---- 102
5. Queues ---- 119
5.1 What is a Queue? ---- 119
5.2 How are Queues Used? ---- 119
5.3 Queue ADT ---- 119
5.4 Exceptions ---- 120
5.5 Applications ---- 120
5.6 Implementation ---- 120
5.7 Problems on Queues ---- 125
6. Trees ---- 135
6.1 What is a Tree? ---- 135
6.2 Glossary ---- 135
6.3 Binary Trees ---- 136
6.4 Types of Binary Trees ---- 137
6.5 Properties of Binary Trees ---- 137

6.6 Binary Tree Traversals ---- 139
6.7 Generic Trees (N-ary Trees) ---- 159
6.8 Threaded Binary Tree Traversals [Stack or Queue less Traversals] ---- 166
6.9 Expression Trees ---- 171
6.10 XOR Trees ---- 174
6.11 Binary Search Trees (BSTs) ---- 174
6.12 Balanced Binary Search Trees ---- 189
6.13 AVL (Adelson-Velskii and Landis) Trees ---- 189
6.14 Other Variations in Trees ---- 206
7. Priority Queues and Heaps ---- 211
7.1 What is a Priority Queue? ---- 211
7.2 Priority Queue ADT ---- 211
7.3 Priority Queue Applications ---- 212
7.4 Priority Queue Implementations ---- 212
7.5 Heaps and Binary Heap ---- 213
7.6 Binary Heaps ---- 214
7.7 Heapsort ---- 218
7.8 Problems on Priority Queues [Heaps] ---- 219
8. Disjoint Sets ADT ---- 232
8.1 Introduction ---- 232
8.2 Equivalence Relations and Equivalence Classes ---- 232
8.3 Disjoint Sets ADT ---- 233
8.4 Applications ---- 233
8.5 Tradeoffs in Implementing Disjoint Sets ADT ---- 233
8.8 Fast UNION implementation (Slow FIND) ---- 234
8.9 Fast UNION implementations (Quick FIND) ---- 237
8.10 Summary ---- 239
8.11 Problems on Disjoint Sets ---- 239
9. Graph Algorithms ---- 241
9.1 Introduction ---- 241
9.2 Glossary ---- 241
9.3 Applications of Graphs ---- 244
9.4 Graph Representation ---- 244
9.5 Graph Traversals ---- 249
9.6 Topological Sort ---- 255
9.7 Shortest Path Algorithms ---- 257
9.8 Minimal Spanning Tree ---- 262
9.9 Problems on Graph Algorithms ---- 266
10. Sorting ---- 286
10.1 What is Sorting? ---- 286
10.2 Why is Sorting necessary? ---- 286
10.3 Classification of Sorting Algorithms ---- 286

10.4 Other Classifications ---- 287
10.5 Bubble sort ---- 287
10.6 Selection Sort ---- 288
10.7 Insertion sort ---- 289
10.8 Shell sort ---- 290
10.9 Merge sort ---- 291
10.10 Heapsort ---- 293
10.11 Quicksort ---- 293
10.12 Tree Sort ---- 295
10.13 Comparison of Sorting Algorithms ---- 295
10.14 Linear Sorting Algorithms ---- 296
10.15 Counting Sort ---- 296
10.16 Bucket sort [or Bin Sort] ---- 296
10.17 Radix sort ---- 297
10.18 Topological Sort ---- 298
10.19 External Sorting ---- 298
10.20 Problems on Sorting ---- 299
11. Searching ---- 309
11.1 What is Searching? ---- 309
11.2 Why do we need Searching? ---- 309
11.3 Types of Searching ---- 309
11.4 Unordered Linear Search ---- 309
11.5 Sorted/Ordered Linear Search ---- 310
11.6 Binary Search ---- 310
11.7 Comparing Basic Searching Algorithms ---- 311
11.8 Symbol Tables and Hashing ---- 311
11.9 String Searching Algorithms ---- 311
11.10 Problems on Searching ---- 311
12. Selection Algorithms [Medians] ---- 333
12.1 What are Selection Algorithms? ---- 333
12.2 Selection by Sorting ---- 333
12.3 Partition-based Selection Algorithm ---- 333
12.4 Linear Selection algorithm - Median of Medians algorithm ---- 333
12.5 Finding the K Smallest Elements in Sorted Order ---- 334
12.6 Problems on Selection Algorithms ---- 334
13. Symbol Tables ---- 343
13.1 Introduction ---- 343
13.2 What are Symbol Tables? ---- 343
13.3 Symbol Table Implementations ---- 343
13.4 Comparison of Symbol Table Implementations ---- 344
14. Hashing ---- 345
14.1 What is Hashing? ---- 345

14.2 Why Hashing? 345
14.3 HashTable ADT 345
14.4 Understanding Hashing 345
14.5 Components of Hashing 346
14.6 Hash Table 347
14.7 Hash Function 347
14.8 Load Factor 348
14.9 Collisions 348
14.10 Collision Resolution Techniques 348
14.11 Separate Chaining 348
14.12 Open Addressing 349
14.13 Comparison of Collision Resolution Techniques 350
14.14 How Hashing Gets O(1) Complexity? 350
14.15 Hashing Techniques 351
14.16 Problems for which Hash Tables are not suitable 351
14.17 Bloom Filters 351
14.18 Problems on Hashing 353
15. String Algorithms 360
15.1 Introduction 360
15.2 String Matching Algorithms 360
15.3 Brute Force Method 360
15.4 Robin-Karp String Matching Algorithm 361
15.5 String Matching with Finite Automata 362
15.6 KMP Algorithm 363
15.7 Boyce-Moore Algorithm 366
15.8 Data Structures for Storing Strings 367
15.9 Hash Tables for Strings 367
15.10 Binary Search Trees for Strings 367
15.11 Tries 367
15.12 Ternary Search Trees 369
15.13 Comparing BSTs, Tries and TSTs 375
15.14 Suffix Trees 375
15.15 Problems on Strings 378
16. Algorithms Design Techniques 386
16.1 Introduction 386
16.2 Classification 386
16.3 Classification by Implementation Method 386
16.4 Classification by Design Method 387
16.5 Other Classifications 388
17. Greedy Algorithms 389
17.1 Introduction 389
17.2 Greedy strategy 389

17.3 Elements of Greedy Algorithms ---- 389
17.4 Does Greedy Always Work? ---- 389
17.5 Advantages and Disadvantages of Greedy Method ---- 390
17.6 Greedy Applications ---- 390
17.7 Understanding Greedy Technique ---- 390
17.8 Problems on Greedy Algorithms ---- 393
18. Divide and Conquer Algorithms ---- 399
18.1 Introduction ---- 399
18.2 What is Divide and Conquer Strategy? ---- 399
18.3 Does Divide and Conquer Always Work? ---- 399
18.4 Divide and Conquer Visualization ---- 399
18.5 Understanding Divide and Conquer ---- 400
18.6 Advantages of Divide and Conquer ---- 400
18.7 Disadvantages of Divide and Conquer ---- 401
18.8 Master Theorem ---- 401
18.9 Divide and Conquer Applications ---- 401
18.10 Problems on Divide and Conquer ---- 401
19. Dynamic Programming ---- 414
19.1 Introduction ---- 414
19.2 What is Dynamic Programming Strategy? ---- 414
19.3 Properties of Dynamic Programming Strategy ---- 414
19.4 Can Dynamic Programming Solve All Problems? ---- 414
19.5 Dynamic Programming Approaches ---- 414
19.6 Examples of Dynamic Programming Algorithms ---- 415
19.7 Understanding Dynamic Programming ---- 415
19.8 Longest Common Subsequence ---- 418
19.9 Problems on Dynamic Programming ---- 420
20. Complexity Classes ---- 451
20.1 Introduction ---- 451
20.2 Polynomial/Exponential time ---- 451
20.3 What is Decision Problem? ---- 451
20.4 Decision Procedure ---- 452
20.5 What is a Complexity Class? ---- 452
20.6 Types of Complexity Classes ---- 452
20.7 Reductions ---- 454
20.8 Problems on Complexity Classes ---- 456
21. Miscellaneous Concepts ---- 459
21.1 Introduction ---- 459
21.2 Hacks on Bitwise Programming ---- 459
21.3 Other Programming Questions ---- 463
References ---- 470

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 artifacts 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 the book are presented in the Python programming language. A unique feature of this book that is missing in most of the available books is to offer a balance between theoretical, practical concepts, problems 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 to practice each exam objective and will help you assess your knowledge before you take the real interview. The detailed answers to every question will help reinforce your knowledge.

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 to 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 and multicore programming.

0.2 Should I Take this Book?

The book is intended for Python programmers who need to learn about algorithmic problem-solving, or who need a refresher. Data and computational scientists employed to do big data analytic analysis should find this book useful. Game programmers and financial analysts/engineers may find this book applicable too. And, students of computer science, or similar programming-related topics, such as bioinformatics, may also find the book to be

quite useful. Although this book is more precise and analytical than many other data structure and algorithm books, it rarely uses any mathematical concepts that are not 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 for their interview preparation.

0.3 Organization of Chapters

Data Structures and Algorithms are important parts of computer science. They form the fundamental building blocks of developing logical solutions to problems. They help in creating efficient programs that perform tasks optimally. This book comprehensively covers the topics required for a thorough understanding of the subjects. It focuses on concepts like 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 in the following way:

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 relationships among various approaches to studying 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 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. Backtracking is a form of recursion. Several 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 *rear* (also called *tail*), and the deletion of exisiting 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 a 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 any database for that matter 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. It can act as an auxiliary data structure for other abstract data types

   It is an example for 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 few of those such as Generic Trees, Binary Trees, Binary Search Trees, and Balanced Binary Trees etc...
7. *Priority Queues*: The *Priority Queue* abstract data type is designed for systems that maintaining a collection of prioritized elements, where elements are removed from the collection in order of their priority. Priority queues turn up in several applications. A simple application comes from 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*. It is an example for 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. Many practical problems can be represented by graphs.
9. *Disjoint Set ADT*: A disjoint sets 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.

   In this chapter, we will limit ourselves 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. Sorting is one of the important categories of algorithms in computer science. Sometimes sorting significantly reduces the complexity of the problem. We can use sorting as a technique to reduce the search complexity. Great research went into this category of algorithms because of its importance. These algorithms are used in many computer algorithms [for example, searching elements], database algorithms and many more. In this chapter, we understand 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 may be elements of other search space.

    Searching is one of core computer science algorithms. We know that today's computers store a lot of information. To retrieve this information proficiently we need very efficient searching algorithms. There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in a 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. In this chapter we will look 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 spell checker. The spell checker is also a dictionary but limited in scope. There are many real time examples for dictionaries and 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 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 *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 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 hashing process and problems which can 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 command line interface (in both Windows and UNIX). After typing the prefix of the directory name if we press tab

button, then we get a list of all matched directory names available. This is another example of auto completion.

In order to support these kind 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 see many algorithms for solving different kinds of problems. Before solving a new problem, the general tendency is to look for the similarity of current problem with other problems for which we have solutions. This helps us in getting the solution easily. In this chapter, we will see different ways of classifying the algorithms and in subsequent chapters we will focus on a few of them (say, Greedy, Divide and Conquer and Dynamic Programming).
17. *Greedy Algorithms*: A greedy algorithm is also be 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 but it is from literature, and 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 as 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/Bitwise 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 few tips and tricks with focus on bitwise operators. Also, it covers few other uncovered and general problems.

At the end of each chapter, a set of problems/questions are 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 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.