

# Python

---

# Quick

---

# Interview

---

# Guide

Top Expert-Led Coding Interview Question Bank for Python Aspirants



SHYAMKANT LIMAYE





# Python Quick Interview Guide

Top Expert-Led Coding Interview Question Bank for Python Aspirants



SHYAMKANT LIMAYE



# **Python Quick Interview Guide**

---

*Top Expert-Led Coding Interview  
Question Bank for Python Aspirants*

---

**Shyamkant Limaye**



[www.bpbonline.com](http://www.bpbonline.com)

**FIRST EDITION 2021**

**Copyright © BPB Publications, India**

**ISBN: 978-93-89423-30-3**

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

#### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

#### **Distributors:**

##### **BPB PUBLICATIONS**

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

##### **MICRO MEDIA**

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

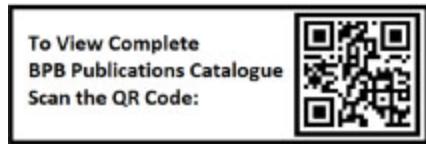
Ph: 22078296/22078297

##### **DECCAN AGENCIES**

4-3-329, Bank Street,  
Hyderabad-500195  
Ph: 24756967/24756400

**BPB BOOK CENTRE**

376 Old Lajpat Rai Market,  
Delhi-110006  
Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

[www.bpbonline.com](http://www.bpbonline.com)

## **Dedicated to**

*Smita, my charming wife,  
without whose silent support this project would never  
have been possible.*

## About the Author

Professor **Shyamkant Limaye** spent 18 years in the computer industry and 30 years in teaching the students of Electronics Engineering. His experience includes a 2-year stint as a System Analyst in the USA. In 1971, he graduated from the Visvesvaraya National Institute of Technology in Electrical engineering with a gold medal. He did his Master's from IIT Kanpur, and is a Doctorate in Electronics from RTM Nagpur University. He has guided 10 students for PhD. He has published a textbook on VHDL programming in 2007. He has also published a thriller novel entitled "Dual reality" in 2011. He retired as a Professor in Electronics and Telecomm Dept. at St. Vincent Pallotti College of Engineering and Technology, Nagpur.

## About the Reviewer

**Dr. Manoj B. Chandak** is head of Computer Science and Engineering at Ramdeobaba College of Engineering and Management, Nagpur. He holds a Ph.D. in Computer Science and Engineering from Nagpur University. He also holds M.E. in Computer science where he was a topper and B.E. in Computer Technology from Nagpur university where he stood second. He is an acknowledged academician with over 22 years of teaching experience with more than 60 research publications in referred journals and conferences. A recognized supervisor in doctoral research in RTM Nagpur University and SGB Amravati University, his research interests are in NLP, Big Data, Information Retrieval, and Mobile & Wireless Technology. He conducts courses in Design and Analysis of Algorithms, Language Processing [Compiler Design], Advances in Algorithms, and Natural Language Processing.

## Acknowledgements

There are a few people I want to thank for helping me to bring this book into existence.

First is Mr. Rohan Vaidya, my young colleague at St. Vincent Pallotti College of Engineering and Technology. He urged me to learn Python because he said that tomorrow's engineers and scientists are going to need it, and as a teacher, it was my duty to teach them. I have been a diehard fan of C++ for the most of my life. But due to Rohan's persuasion, I decided to learn Python. Interestingly, it was my B.E. final year student, Pravesh Bawangade, who gave me the first lesson of Python. I found the language to be really useful.

After interacting with students going for campus interviews, I realized that there are many introductory books on Python and many advanced level books dealing with the data science and artificial intelligence. But, there is no book that grills them into coding and teaches algorithms and data structures from a competition point of view. I, therefore, decided to write this book.

I am thankful to my colleagues Dr. Ms. Hema Kale, Dr. Manoj Chandak, and Dr. Rajesh Pande, for going through my early drafts and giving valuable suggestions.

Finally, I must thank BPB Publications for accepting my idea and giving continuous support during the editing process.

# Preface

It gives me great pleasure to offer this book to the students. C++ was invented in 1979 and Python was invented 10 years later, in 1989. C++, and its cousin Java, ruled the programming world for many years. However, they are loosening their grip now. Python offers more flexible language constructs and platform independence. But the recent upsurge in the demand for Python is triggered by a rich set of libraries in the public domain, especially in the area of artificial intelligence (AI).

Many excellent books have been written on the Python programming. Some are basic and some teach advanced topics, like, machine learning and deep learning. But to get into a good company, firstly, a student has to clear an aptitude test and a coding test. While many books exist on the aptitude preparation, there is a dearth of books to prepare the students for a coding interview. The standard university curriculum does not provide this knowledge. The coding test can be given in many languages like C, C++, or Java. However, since Python is the language of the future, it is better to give it in Python.

There are some resources available on the internet, but they are scattered over many places, and they neither provide proper explanation, nor provide a graded path from easy to hard problems. Moreover, one cannot become a coding wizard merely by copy-pasting somebody's code without understanding. That is precisely why this book has been written. In addition, a college may choose this book as a reference book for the lab courses in Python.

It is not possible to cover all the possible problems in a small book. But it is **neither necessary, nor desirable**. The 75 problems presented in this book will guide the students to the path of self-reliance and confidence.

Every year, the placement agencies and IT companies conduct coding tests for the graduating batch. Many of them use an artificial intelligence-based evaluation tool that senses the logic of the students. If you get the answer right, it is good. But, even if you don't get a valid answer, or cannot remove

the compilation errors, or cannot remove the run time errors, the tool is able to sense your logic. It is also able to sense the time complexity and the space complexity of the code. Most companies in India, and abroad, use similar sets of questions and similar evaluation tools. In general, **the students' performance in coding tests has been found to be dismal**. Without coding skills, you cannot enter into well-paying companies.

As a teacher, I am naturally worried. We teach you C and C++, you do practicals and pass the university exams. But still, the recruiter says you are not up to the mark. After a lot of interaction with the placement officers of various colleges and past students, I found out that you need additional practice in coding. The FAANG companies (Short for Facebook, Amazon, Apple, Netflix, Google) are the world's best paymasters. Some people add Microsoft to that list. All of them make you take coding tests. For clearing these tests, you need two things.

1. A good knowledge of algorithmic strategies and data structures. This enables you to think about the solution.
2. A good knowledge of the programming language which will implement your thought.

### **Shortcomings of the university syllabus**

A typical university syllabus covers the basic language syntax and a few well-known examples in the data structures. Cracking a coding test requires a much more rigorous study. You need to understand the problem, formulate the solution strategy, optimize the solution, and successfully execute it online. Even if you are unable to run the code due to some syntax errors, don't worry. The evaluation tool is smart enough to understand your logic and give stepwise marks. The first step is to get the correct output. You may use a simple but inefficient method, to begin with. Such methods are called **Brute force** methods. Don't underestimate them. They do fetch you the credit. But for getting a better rank, you must understand the time complexity, the space complexity, and the big O notation, for example,  $O(n^2)$ . You should also know the tricks to reduce the time complexity. You should know the strategies like hash table, sliding window, binary search, dynamic programming, depth-first search, breadth-first search, and so on. Some companies take a *whiteboard test* instead of an online coding test.

Here, they ask you to go to a whiteboard and write your program. Some people may ask you to write on a paper or sheet as well. Such tests are not conducted in a typical university practical examination.

## **Removing the shortcomings**

Well, you might ask, “How do I prepare myself for this test?” There are several resources available on the internet for this, like LeetCode, HackerRank, Topcoder, Codeforces, GeeksforGeeks, CodeChef, SPOJ, HackerEarth, and so on. While the above sources and so many others on the internet are great, going through the net is very time-consuming. In the beginning, even the easy problems will feel hard to you. So, you need a consolidated book that will hold your hand and guide you through the steps. You also need a set of solved problems and a methodology to start thinking about the solution strategy. Then, you can gradually shift from easy to hard problems on your own. This is exactly why I have written this book. It straightway jumps to coding examples. The theory is introduced wherever needed. After practicing these problems, you will be ready to enter the global challenges and get top offers. The book exposes you to advanced algorithmic tactics, like binary search, recursion, divide-and-conquer, dynamic programming, memoisation, greedy approach, and so on.

Python and its libraries are like an ocean. There is no limit to how deep you can dive into it. You can learn packages like, sklearn and TensorFlow for machine learning, or OpenCV for image processing, or so many other things. You can impress the interviewer with your projects on these topics, – **IF** – you reach the interview stage. And that’s a big **IF**. For reaching that stage, what you need is the basic coding skills – with no packages. So, first things first. Let us become a coding wizard.

## **Can I do it?**

Some important questions you might be pondering might be: Can I really crack the FAANG tests? Am I talented enough? The answer is a resounding YES. Many of my students, who were not recognized to be academically bright, developed great coding skills through their efforts and landed in plush jobs. **Who knows, you may be having a hidden talent that you did not know.**

You may think, “The tests for highly-paid jobs must be tough; there is no chance I will get a good rank in those. Then, all the efforts I take will go waste”. Don’t think like that. The outcome of the coding test is not binary – 0 or 1 – FAANG or unemployed. Let me assure you that the industry offers a wide continuum of jobs depending on the level of your preparation. I agree that FAANG is a bit competitive, but there are many other great opportunities, if you are reasonably good.

## Prerequisites

The book assumes that you have a basic knowledge of Python, data structures, and algorithms. In short, you have passed some university examination in C/C++ and done an introductory course in Python. Though installing Python is theoretically not necessary, because you can test your code on several online executors, I recommend that you should install a Python IDE with step-by-step debugging capability on your computer. Spyder or PyCharm are ideal. The examples in this book have been tested on Python 3.7.

**Finally, if you are a student of a core branch, like, ETC or mechanical, what good is this preparation?** This preparation will take 60-80 hours, not a very great amount. Preparation never goes waste. It will give you a very good analytical thinking ability that you can use in your own field. Moreover, you have a backup option for jobs.

This book presents 75 frequently asked coding questions. It discusses solution strategy and provides working code. It equips you with the skills required for developing and analyzing algorithms for various situations. It is, by no means, exhaustive, but it is sufficient to make you confident and capable of exploring further on your own.

Over the 9 chapters in this book, you will learn the following:

[\*\*Chapter 1\*\*](#) explains the concepts of the time complexity and space complexity, and presents practical techniques to measure them. Then, it presents questions based on binary search, lists, and strings.

[\*\*Chapter 2\*\*](#) presents questions on the linked lists and stacks.

[\*\*Chapter 3\*\*](#) presents questions on hash tables and maths.

[\*\*Chapter 4\*\*](#) presents questions on trees and graphs.

[\*\*Chapter 5\*\*](#) presents questions on depth-first search.

[\*\*Chapter 6\*\*](#) presents questions on breadth-first search.

[\*\*Chapter 7\*\*](#) presents questions on backtracking.

[\*\*Chapter 8\*\*](#) presents questions on the greedy and divide-and-conquer algorithms.

[\*\*Chapter 9\*\*](#) presents questions on dynamic programming.

## **Downloading the code bundle and coloured images:**

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

**<https://rebrand.ly/umopksr>**

## **Errata**

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**[errata@bpbonline.com](mailto:errata@bpbonline.com)**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at **[www.bpbonline.com](http://www.bpbonline.com)** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **[business@bpbonline.com](mailto:business@bpbonline.com)** for more details.

At **[www.bpbonline.com](http://www.bpbonline.com)**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.



## **BPB is searching for authors like you**

If you're interested in becoming an author for BPB, please visit [www.bpbonline.com](http://www.bpbonline.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Python-Quick-Interview-Guide>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/bpbpublications>. Check them out!

## **PIRACY**

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [business@bpbonline.com](mailto:business@bpbonline.com) with a link to the material.

## **If you are interested in becoming an author**

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit [www.bpbonline.com](http://www.bpbonline.com).

## **REVIEWS**

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase

decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).

# Table of Contents

## 1. Lists, Binary Search, and Strings

Structure

Objectives

### 1.1 Time and Space Complexity.

1.1.1  $O(n)$ .

1.1.2  $O(1)$ .

1.1.3  $O(n^2)$ .

1.1.4  $O(n^3)$ .

1.1.5  $O(n \log n)$ .

1.1.6  $O(2^n)$ .

1.1.7 Space complexity.

### 1.2 Linear Data Structures in Python

1.2.1 The list class

1.2.2 NumPy arrays

1.2.3 Strings

### 1.3 Sorting and Searching

Searching

#### 1.4 Question 1: What is the position of a target in a given sorted array?

Problem statement

Solution format

Strategy

Approach 1 – Linear Search

Python code

Complexity Analysis

Approach 2 – Binary Search

Python code

Complexity Analysis

#### 1.5 Question 2 – Is the given integer a valid square?

Problem statement

Solution format

Strategy

[Approach 1: Linear Search](#)

[Python code](#)

[Complexity Analysis](#)

[Approach 2 – Summation of Arithmetic Progression](#)

[Python code](#)

[Complexity Analysis](#)

[1.6 Question 3 – How will you move zeroes in a given array to the end?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Pseudo code](#)

[Python code](#)

[Complexity Analysis](#)

[1.7 Question 4 – How many boats are required to save people?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

[1.8 Question 5 – Is the given array a valid mountain array?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

[1.9 Question 6 – Which container can store maximum water?](#)

[Problem statement](#)

[Strategy](#)

[Brute force approach](#)

[Pinching window approach](#)

[Answer format](#)

[Python code](#)

[Complexity Analysis](#)

[1.10 Question 7 – Which was the first faulty version of the software?](#)

[Problem statement](#)

[Solution format](#)

Strategy

Python code

Complexity Analysis

## 1.11 Question 8 – What are all the subsets of a given set of integers?

Problem statement

Solution format

Strategy

Approach 1 – Growing from seeds

Python code

Complexity Analysis

Approach 2: Binary Representation

Python code

Complexity Analysis

Python code

Complexity Analysis

## 1.12 Background – Measuring the execution time and the time complexity

1.12.1 Introduction

1.12.2 Using perf\_counter

1.12.3 Use of timeit

1.12.4 Plotting the time complexity graphs

Python code

Plotting a graph for the sort function

Python code

Plotting a graph for the subsets of a set problem

Python code

## 1.13 Question 9 – What is the maximum sum of a subarray of a given array?

Problem statement

Solution format

Strategy

Brute force method

Python code

Complexity Analysis

Python code

Efficient method

[Python code](#)

[Python code](#)

[1.14 Question 10 – What is the integer part of the square root of a given integer?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Approach 1 – Linear search](#)

[Python code](#)

[Complexity Analysis](#)

[Approach 2 – Binary search](#)

[Python code](#)

[Complexity Analysis](#)

[1.15 Question 11 – What are the first and last positions of a target number in a given sorted array?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Linear search \(Brute force\)](#)

[Python code](#)

[Complexity Analysis](#)

[Binary search](#)

[Python code](#)

[Python code](#)

[Complexity Analysis](#)

[1.16 Question 12 – What is the position of a search target in a 2D matrix?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Approach 1 – Linear search](#)

[Complexity Analysis](#)

[Approach 2 – 2D Linear search](#)

[Python code](#)

[Complexity Analysis](#)

[Approach 3 – 2D Binary search](#)

[Python code](#)

[Complexity Analysis](#)

[1.17 Question 13 – How will you convert an integer into a roman numeral?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

[1.18 Question 14 – How will you construct Pascal’s triangle?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

[Conclusion](#)

[Points to Remember](#)

[MCQs](#)

[Answers to MCQs](#)

[Questions](#)

[Key Terms](#)

## [2. Linked Lists and Stacks](#)

[Structure](#)

[Objectives](#)

[2.1 Basics of Linked Lists](#)

[2.1.1 Node, the basic building block](#)

[2.1.2 Creating and displaying a linked list](#)

[Python code](#)

[2.1.3 Adding a node in the beginning](#)

[Python code](#)

[2.1.4 Adding a node at the end](#)

[Python code](#)

[2.1.5 Creating a list from an array](#)

[Python code](#)

## 2.2 Question 15 – How will you merge two sorted lists?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 2.3 Question 16 – How will you detect a cycle in a linked list?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 2.4 Question 17 – How will you reverse a linked list?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 2.5 Question 18 – How will you add two linked lists?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 2.6 Question 19 – How will you remove the $n^{\text{th}}$ node from the right?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 2.7 Question 20 – How will you create odd and even linked lists?

Problem statement

Solution format

Strategy

Python code

## Complexity Analysis

2.8 Question 21 – How will you evaluate the reverse polish postfix expression?

Problem statement

Solution format

Strategy

Python code

## Complexity Analysis

2.9 Question 22 – How will you achieve the minimum implementation of a stack using a list?

Problem statement

Solution format

Strategy

Approach 1:  $O(n)$

Python code

Approach 2 –  $O(1)$

Python code

## Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to MCQs

Questions

Key terms

## **3. Hash Table and Maths**

Structure

Objectives

3.1 Implementation of Hash Table in Python

Using “`defaultdict`”

Using “`set`”

3.2 Question 23 – How will you find if an array contains duplicates?

Problem statement

Solution format

Strategy

Approach 1 – Brute force

Python code

Complexity Analysis

Approach 2 – Sorting

Python code

Complexity Analysis

Approach 3 – Using the “set” method

Python code

Complexity Analysis

Approach 4: Using hash tables

Python code

Complexity Analysis

3.3 Question 24 – How will you find if an array contains duplicates in the vicinity?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

3.4 Question 25 – How will you find the majority element in an array?

Problem statement

Solution format

Strategy

Approach 1 – From basics

Python code

Approach 2 – Using the “count” method of the list class

Python code

Approach 3 – Using the “get” method of the dictionary with default

Python code

Approach 4 – Using the “Counter” method from the collections module

Python code

Complexity Analysis

3.5 Question 26 – How will you find if the string of brackets is valid?

Problem statement

Solution format

Strategy

Pseudo code

Python code

Complexity Analysis

3.6 Question 27 – How will you find two numbers whose sum equals the target?

Problem statement

Solution format

Strategy

Approach 1 – Brute force method

Python code

Complexity Analysis

Approach 2 – Hash table

Python code

Complexity Analysis

3.7 Question 28 – How will you count the primes less than  $n$ ?

Problem statement

Solution format

Strategy

Approach 1 – Brute force method

Python code

Complexity Analysis

Approach 2 – Eratosthenes sieve method

Python code

Complexity Analysis

3.7 Question 29 – How will you find the longest substring without the repeating characters?

Problem statement

Solution format

Strategy

Approach 1 – Brute force

Python code

Complexity Analysis

Approach 2 – Sliding window

Python code

## Complexity Analysis

3.8 Question 30 – How will you convert a roman numeral into a decimal numeral?

Problem statement

Solution format

Strategy

Python code

## Complexity Analysis

3.9 Question 31 – How will you identify a single number in an array?

Problem statement

Solution format

Strategy

Approach 1 – Using a list

Python code

## Complexity Analysis

Approach 2 – Using a hash table

Python code

## Complexity Analysis

Approach 3 – Algebraic method

Python code

## Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to the MCQs

Questions

Key terms

## **4. Trees and Graphs**

Structure

Objectives

4.1 Basics of Graph Theory

4.2 Representation and Manipulation of Graphs

4.2.1 Representation

4.2.2 Manipulation of graphs

Python code

#### 4.3 Representation and Manipulation of Trees

4.3.1 Representation of binary trees in memory

4.3.2 Creation and display

Python code

Python code

Python code

#### 4.4 Recursion

##### 4.5 Question 32 – How will you detect a redundant road connection?

Problem statement

Solution format

Strategy

Python code

Complexity

##### 4.6 Question 33 – How will you find the lowest common ancestor in a binary tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

##### 4.7 Question 34 – Who is the town judge?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

##### 4.8 Question 35 – How will you select flowers so that the adjacent gardens are not the same?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

Conclusion

Points to remember

[MCQs](#)

[Answers to MCQs](#)

[Questions](#)

[Key terms](#)

## [5. Depth First Search](#)

[Structure](#)

[Objectives](#)

### [5.1 DFS Traversal of Graphs and Trees](#)

[Python code](#)

[Python code](#)

[Complexity Analysis](#)

[DFS Traversal of Trees](#)

### [5.2 Question 36 – How will you reconstruct the itinerary from tickets?](#)

[Problem statement](#)

[Solution format:](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

### [5.3 Question 37 – How will you validate a symmetric binary tree?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Approach 1 – Recursive](#)

[Python code](#)

[Complexity Analysis](#)

[Approach 2 – Iterative](#)

[Python code](#)

[Complexity Analysis](#)

### [5.4 Question 38 – How will you find the maximum height of a binary tree?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

### Complexity Analysis

#### 5.5 Question 39 – How will you find the path sum in a binary tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 5.6 Question 40 – What is the $K^{\text{th}}$ smallest element in a binary search tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis:

#### 5.7 Question 41 – How will you find the maximum path sum in a binary tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 5.8 Question 42 – How will you validate a balanced binary tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 5.9 Question 43 – How will you validate a binary search tree?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 5.10 Question 44 – What is the number of islands?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

### 5.11 Question 45 – How will you remove the surrounded islands?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

### 5.12 Question 46 – Is the course schedule valid?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

### 5.13 Question 47 – How will you reward a sales manager?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to the MCQs

Questions

Key Terms

## **6. Breadth First Search**

Structure

Objectives

### 6.1 Basic BFS graph class

Python code

Recursive approach

Python code

## 6.2 Question 48 – How will you print a tree level-wise?

Problem statement

Solution format

Strategy

Python code

Complexity

## 6.3 Question 49 – How will you build a word ladder between two words by changing one letter at a time?

Problem statement

Solution format

Strategy

Python code

Complexity

## 6.4 Question 50 – How will you find a course sequence according to pre-requisites?

Problem statement

Solution format

Strategy

Python code

Complexity

## 6.5 Question 51 – How will you test if two nodes of a tree are cousins?

Problem statement

Solution format

Strategy

Python code

Complexity

## 6.6 Question 52 – How will you find the shortest bridge across two islands?

Problem statement

Solution format

Strategy

Python code

Complexity

Conclusion

Points to remember

[MCQs](#)

[Answers to MCQs](#)

[Questions](#)

[Key terms](#)

## [7. Backtracking](#)

[Structure](#)

[Objectives](#)

### [7.1 Backtracking principle](#)

[Some famous backtracking problems](#)

[Place N-Queens](#)

[Find a Hamiltonian circuit in the graph](#)

### [7.2 Question 53 – How will you search a word in a grid?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

### [7.3 Question 54 – How will you traverse a maze?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

### [7.4 Question 55 – How will you find all the combinations of \$n\$ numbers](#)

[taken  \$k\$  at a time?](#)

[Problem statement](#)

[Solution format](#)

[Strategy](#)

[Python code](#)

[Complexity Analysis](#)

### [7.5 Question 56 – How will you partition a string into palindrome](#)

[segments?](#)

[Problem statement](#)

[Solution format](#)

Strategy

Python code

Complexity Analysis

7.6 Question 57 – What is the sum of the elements of the BST within a range?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

7.7 Question 58 – How will you partition a set into  $k$  subsets having equal sum?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to MCQs

Questions

Key terms

## 8. Greedy and Divide-and-Conquer Algorithms

Structure

Objectives

8.1 Some famous problems on greedy methodology

8.2 Some famous applications of divide-and-conquer strategy

8.3 Question 59 – How will you maximize the value in a knapsack?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

8.4 Question 60 – How will you remove  $k$  digits from a number to get the smallest number?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

8.5 Question 61 – Can you provide the correct change to lemonade customers?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

8.6 Question 62 – How will you use the divide-and-conquer strategy for sorting?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

8.7 Question 63 – How will you find the  $k$  closest points to the origin?

Problem statement

Solution format

Strategy

Approach 1 – Sorting

Python code

Complexity Analysis

Approach 2 – Selection sort

Python code

Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to the MCQs

Questions

## Key terms

### 9. Dynamic Programming

Structure

Objectives

#### 9.1 Principles of dynamic programming

Problem statement

Python code

Python code

Problem statement

Strategy

Python code

#### 9.2 Question 64 – How will you maximize the value in a 0/1 knapsack?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 9.3 Question 65 – How to maximize the sales of a door-to-door

salesman?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

#### 9.4 Question 66 – What is the best time to buy and sell stock?

Problem statement

Solution format

Strategy

Approach 1 – Brute force

Python code

Complexity Analysis

Approach 2 – Dynamic programming

Python code

Complexity Analysis

## 9.5 Question 67 – What is the best time to buy and sell stock, part 2?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 9.6 Question 68 – In how many ways can you climb the stairs?

Problem statement

Solution format

Strategy

Approach 1 – Pure dynamic programming

Python code

Complexity Analysis

Approach 2 – Compact array

Python code

Complexity Analysis

Approach 3 – Matrix exponentiation

Python code

Python code

Complexity Analysis

Approach 4 – Binet’s formula

Python code

Complexity Analysis

## 9.7 Question 69 – How will you minimize the cost of climbing stairs?

Problem statement

Solution format

Strategy

Consider i = 1

Consider i = 2

Python code

Complexity Analysis

## 9.8 Question 70 – What is the minimum number of coins to dispense the change?

Problem statement

Solution format

Strategy

Consider i = 1

Consider i = 2

Consider i = 3

Consider i = 4

Consider i = 5

Python code

Python code

Complexity Analysis

## 9.9 Question 71 – How many coin patterns to dispense the change?

Problem statement

Solution format

Strategy

Consider i= first coin and value = 1

Now consider i=2

Now consider i=5

Consider i = 2 (Only one value).

Python code

Complexity Analysis

## 9.10 Question 72 – How many unique paths exist in a square grid?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 9.11 Question 73 – How many unique paths exist in a square grid

having obstacles?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

## 9.12 Question 74 – What is the longest palindromic substring of a given string?

Problem statement

Solution format

Strategy

Consider  $j=1$

Consider  $j=2$

Consider  $j=3$

Consider  $j=4$

Python code

Complexity Analysis

## 9.13 Question 75 – How much rain water can be trapped in the ridges?

Problem statement

Solution format

Strategy

Python code

Complexity Analysis

Conclusion

Points to Remember

MCQs

Answers to MCQs

Questions

Key terms

**Index**

# CHAPTER 1

## Lists, Binary Search, and Strings

Most of the coding interview questions are based on lists, binary search, and strings. In this chapter, we will solve many problems on these. If you have brushed up your knowledge on these topics, it is good. But even if you haven't, don't worry we will give you tips as we proceed.

### Structure

The concepts of time complexity and space complexity are very important in competitive coding. We will introduce these concepts first and do some warm up exercises during that process. After you have mastered a few problems, we will take a little diversion and present a wonderful topic on practical measurement and the plotting of time complexity.

Then, we will end the chapter with some more problems. In the course of solving these problems, you will be introduced to binary search and sliding window algorithms. We will cover the topics in the following order:

- Time and space complexity
- Linear data structures in Python
- Searching and sorting
- Question 1: What is the position of a target in a given sorted array?
- Question 2: Is the given integer a valid square?
- Question 3: How will you move zeroes in a given array to the end?
- Question 4: How many boats are required to save people?
- Question 5: Is the given array a valid mountain array?
- Question 6: Which container can store maximum water?
- Question 7: Which was the first faulty version of the software?
- Question 8: What are all the subsets of a given set of integers?
- Background: Measuring execution time and time complexity

- Question 9: What is the maximum sum of a sub-array of a given array?
- Question 10: What is the integer part of the square root of a given integer?
- Question 11: What are the first and last positions of a target number in a given sorted array?
- Question 12: What is the position of a search target in a 2D matrix?
- Question 13: How will you convert an integer into a roman numeral?
- Question 14: How will you construct Pascal's triangle?

## Objectives

After studying this chapter, you will be able to understand the concepts of time and space complexities; practically measure time complexity; acquire the skills to analyze and solve questions on lists, binary search, and strings

### 1.1 Time and Space Complexity

There are always different approaches to solve a problem. All of them produce the same result, but some are smart, that is, they consume fewer resources, like time and memory. We need to find the approaches that are smart. When we execute a student program on a PC, we don't bother how long it takes to execute it or how much memory it consumes. We always find the computer memory to be big enough for us and also that we get the answer in a few seconds. But real-life problems are bigger. They will test the limitations of the hardware. Therefore, we should be interested in knowing how the execution time of an algorithm behaves when the size of the input –  $n$ , becomes arbitrarily large. A practical thumb rule is if the execution time is proportional to  $n$ , then a typical computer can solve a problem of size  $n = 1$  million in a reasonable time. But, if the execution time is proportional to  $n^2$ , then the reasonable size reduces to  $n = 1000$ , and if the execution time is proportional to  $2^n$ , then the reasonable size reduces to  $n = 20$  only.

Suppose a function  $f(n)$ , describes the relationship between the execution time and the size of the input to be processed. We want to find a simplified function  $g(n)$ , which can compare the performance of various approaches. The function should be independent of the CPU speed and it should estimate

the order of growth of the execution time for large values of  $n$ . Such estimates are called asymptotic estimates. But the execution time does not depend on  $n$  alone. It also depends on the nature of the data. For example, if the input to a Bubble Sort algorithm is already sorted, then the function just makes one pass through the data and finishes. The time required is proportional to  $n$ . But if the input data is sorted in the reverse order, then the function has to do a lot of work in rearranging it. The time required here is proportional to  $n^2$ . A typical input will be partially ordered. Thus, there is a best case timing, a worst case timing, and an average case timing for any algorithm based on the nature of the input data. To represent these cases, we use the big  $\Omega$  notation, big  $O$  notation, and big  $\theta$  notation respectively. The same notations are also used to describe the relationship between the memory requirement and  $n$ . Sometimes we can trade in space for getting a better time.

To understand the asymptotic behavior, consider that the worst case time estimate of an algorithm to run on one computer is described by the following relation:

$$f(n) = 2n^2 + 6n + 100 \text{ microseconds}$$

If we run the same algorithm on another computer which is half as fast, the timing relationship will be as follows:

$$f_2(n) = 4n^2 + 12n + 200 \text{ microseconds}$$

Let us define a simplified function  $g(n)$ , and a constant  $c$ , such that  $c * g(n)$  is greater than  $f(n)$  for large values of  $n$ . Actually, we could find a number  $n_0$  such that if  $n > n_0$ ,  $c * g(n) > f(n)$ . We can get such a function by ignoring the lower order terms of  $f(n)$  and the constant multiplier of the highest order term. Thus,  $g(n) = n^2$  and the constant  $c$  will be as follows:

$$c > \lim_{n \rightarrow \infty} f(n)/g(n)$$

$n \rightarrow \infty$

Thus,  $c$  can be chosen as 3. For small values of  $n$ ,  $g(n)$  is less than  $f(n)$ . For example, for  $n = 1$ ,  $c * g(n) = 3$ , and  $f(n) = 108$ . But we are interested in large values. As  $n$  grows,  $c * g(n)$  catches up. For  $n > 7$ ,  $3g(n)$  is greater than  $f(n)$ . In Big O notation, we write the following:

$$\text{Worst case time complexity} = O(g(n)) = O(n^2).$$

If  $f(n)$  represents the best case values, then, in Big  $\Omega$  notation, we write the following:

*Best case time complexity =  $\Omega(n^2)$ .*

If  $f(n)$  represents the average values, then, in Big  $\theta$  notation, we write the following:

*Average case time complexity =  $\theta(n^2)$ .*

In practice, we are generally interested in estimating the worst case performance. So we will concentrate on the big O notation.

Let us take a look at the complexity function  $g(n)$ .

### 1.1.1 O(n)

$O(n)$  means the execution time of a program varies linearly with the size of the data. For example, say we want to find the sum of all the elements in an array. Then the execution time will be proportional to the number of elements in the array. If  $Ta$  is the additional time, then, the total time  $T$  is given by:

$$T = Ta \ n$$

We are generally not interested in the constant  $Ta$  as it depends on the clock speed of the CPU, word length of the CPU, compiler, operating system, and so many other things. So, we drop the constant  $Ta$  and just say that the time complexity is  $O(n)$ .

Here are some examples of the  $O(n)$  time complexity:

- A program that calculates the dot product of two arrays, that is, element by element multiplication.
- Searching an item in an unsorted array (linear search).

Note that searching an unsorted array will not always take  $n$  operations. It may be anywhere between 1 to  $n$ . But for the  $O(n)$  calculations, we always consider the worst case scenario.

### 1.1.2 O(1)

If the algorithm takes the constant  $K$  seconds, irrespective of the size of the input, its time complexity is  $O(1)$ . For example, the time required to insert a new item at the head of the list is constant. It does not depend on the length of the list. This does not mean that an  $O(1)$  operation is always faster. This just means that it is independent of the input size. For example, consider searching an item in a hash table (which is called a dictionary in Python). The time required for 1 search is independent of the size of the table, so we will describe it as  $O(k)$ . However, it needs to spend a lot of time on the hashing function that computes the address. It is quite high compared to a simple retrieval from an array. For a small number of items, a linear search might even be faster than a hash table search. Beyond a certain size, the hash table will be faster. As we can drop the constants, the time complexity of the algorithm is said to be  $O(1)$ , rather than  $O(k)$ .

### 1.1.3 $O(n^2)$

Now, suppose we want to find the sum of all the elements of an  $n \times n$  matrix. The Python code for this is as follows:

**Note: If you paste the code in the Python interpreter, remove the line numbers.**

```
1. a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
2. sum = 0  
3. for i in range(3):  
4.     for j in range(3):  
5.         sum += a[i][j]  
6. print(sum)
```

**Output:** 45

Refer to the following screenshot:

The screenshot shows the Spyder Python IDE interface. On the left, the code editor displays a file named 'on2.py' with the following content:

```

1 a = [[1,2,3],[4,5,6],[7,8,9]]
2 sum = 0
3 for i in range(3):
4     for j in range(3):
5         sum += a[i][j]
6 print(sum)
7

```

On the right, the Variable explorer shows the state of variables:

Name	Type	Size	Value
a	list	3	[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
i	int	1	2
j	int	1	2
sum	int	1	45

The IPython console below shows the output of the code execution:

```

In [8]: runfile('C:/Users/User/.spyder-py3/on2.py', wdir='C:/Users/User/.spyder-py3')
45

In [9]:

```

At the bottom, the status bar indicates: conda: base (Python 3.7.6) Line 2, Col 8 ASCII CRLF RW Mem 66%

**Figure 1.1:** Screenshot for the example of  $O(n^2)$

Whenever we see two nested loops, we can conclude that there will be  $n^2$  iterations. We will require  $n^2$  additions. Assume that  $K$  is the initial set up of the time needed for creating the matrix and initializing the sum to 0. The execution time  $T$  will be as follows:

$$T = n^2 Ta + K$$

Remember that we are interested in finding the run time when  $n$  becomes very large. Therefore, when the run time dependency is expressed as a polynomial in  $n$ , we are interested only in the highest power of  $n$ . So we can drop the second term from the preceding expression:

$$T = n^2 Ta$$

As stated above, the constant  $Ta$  can also be dropped. The time complexity is simply  $O(n^2)$ .

Similarly, in many cases, we get the time complexity as  $O(n(n-1)/2)$ . It is simplified as  $O(n^2)$ .

## 1.1.4 $O(n^3)$

Now, consider the multiplication of two matrices. First, we will write a *C* style program for this so that we can understand the complexity. Later, we will see the Python shortcuts. In Python, a matrix is created as a list of lists. Let us create  $3 \times 3$  matrices,  $X$  and  $Y$ . We also need a result matrix to hold the result. Then, we will write two nested loops to cover all the row and column indices, and a third loop to calculate the dot product of the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column. The Python code to do this is as follows:

```
1. # Program to multiply two matrices using nested loops
2. # Create 3x3 matrix X
3. X = [[9,7,3],
4.       [4,1,6],
5.       [7,8,9]]
6. # Create 3x3 matrix Y
7. Y = [[5,8,1],
8.       [6,1,3],
9.       [4,5,9]]
10. # Create empty result matrix
11.
12. result = [[0,0,0],
13.             [0,0,0],
14.             [0,0,0]]
15.
16. for i in range(len(X)):
17.     # iterate through columns of Y
18.     for j in range(len(Y[0])):
19.         # iterate through rows of Y
20.         for k in range(len(Y)):
21.             result[i][j] += X[i][k] * Y[k][j]
22. print(result)
```

### **Output:**

```
[[99, 94, 57], [50, 63, 61], [119, 109, 112]]
```

Refer to the following screenshot:

The screenshot shows the Spyder Python 3.7 IDE interface. On the left, the code editor displays a script named 'on3.py' containing Python code for multiplying two 3x3 matrices using nested loops. The code defines matrices X and Y, initializes a result matrix, and iterates through all elements to calculate the product. On the right, the variable explorer shows the state of variables: X, Y, i, j, k, and result. The result matrix is a list of lists: [[9, 7, 3], [4, 1, 6], [7, 8, 9]] and [[5, 8, 1], [6, 1, 3], [4, 5, 9]]. The result of the multiplication is shown in the IPython console as a list of lists: [[99, 94, 57], [50, 63, 61], [119, 109, 112]].

**Figure 1.2:** Screenshot for the example of  $O(n^3)$

The result is correct but not neatly formatted in a matrix form. Remember that the result is a list of lists. We can print each list separately to get a neat print. Replace the last line with these two lines:

```
for r in result:  
    print(r)
```

The output is as follows:

```
[99, 94, 57]  
[50, 63, 61]  
[119, 109, 112]
```

Due to the three nested loops, the innermost statement is executed  $n^3$  times. The time complexity of a matrix multiplication is therefore  $O(n^3)$ .

Now, let us take the advantage of the NumPy package. The simplified code is as follows:

1. **import** numpy as np
2. x = np.array([[9, 7, 3],
3.               [4, 1, 6],

```

4.                 [7,8,9]])
5. y = np.array([[5,8,1],
6.                 [6,1,3],
7.                 [4,5,9]])
8. result = np.matmul(x, y)
9. print(result)

```

## Output:

```

[[99, 94, 57]
[50, 63, 61]
[119, 109, 112]]

```

Refer to the following screenshot:

The screenshot shows the Spyder Python IDE interface. The top menu bar includes File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, and Help. The title bar says "Spyder (Python 3.7)". The left sidebar shows a file tree with "on3numpy.py" selected. The main code editor window displays the following Python code:

```

1 import numpy as np
2 x = np.array([[9,7,3],
3                 [4,1,6],
4                 [7,8,9]])
5 y = np.array([[5,8,1],
6                 [6,1,3],
7                 [4,5,9]])
8 result = np.matmul(x, y)
9 print(result)
10

```

To the right of the code editor is a "Variable explorer" window showing the state of variables:

Name	Type	Size	Value
result	Array of int32	(3, 3)	<pre>[[ 99  94  57]  [ 50  63  61]  [119 109 112]]</pre>
x	Array of int32	(3, 3)	<pre>[[9 7 3]  [4 1 6]  [7 8 9]]</pre>
y	Array of int32	(3, 3)	<pre>[[5 8 1]  [6 1 3]  [4 5 9]]</pre>

Below the variable explorer is a "Console 1/A" window showing the execution history:

```

In [10]: runfile('C:/Users/User/.spyder-py3/on3numpy.py',
      wdir='C:/Users/User/.spyder-py3')
[[ 99  94  57]
 [ 50  63  61]
 [119 109 112]]

In [11]:

```

At the bottom of the interface, status bars show "conda: base (Python 3.7.6)" and "Line 1, Col 1".

**Figure 1.3:** Screenshot for the example of  $O(n^3)$  using NumPy

The time complexity is still  $O(n^3)$ , but the details are hidden in the library implementation.

**Note:** Make sure to use the `np.matmul` function to multiply the matrices. If you use a simple `*` operator, it results in the element by element multiplication.

### 1.1.5 O(n log n)

Now, consider that we are given an array of integers sorted in ascending order.

For example,  $a = [-4, -1, 3, 7, 13, 2, 5]$

We want to find the position (index) of the number 13 in the array. If the number does not exist, the program should return -1. There are two well-known approaches to this, namely linear search and binary search.

In the linear search method, we examine each element of the array and see if it is 13. If a match is found, we return its index. In the best case, the target may be found in the first index. In the worst case, we will do  $n$  compare operations. For big O notation, we always consider the worst case. So, the time complexity is  $O(n)$ .

The binary search method is more efficient, but it requires the array to be sorted. Here, we create the left and right pointers initially set to 0 and  $n-1$ . Then, we compute  $mid = (left + right)/2$ , and examine  $a[mid]$ . If  $a[mid] = 13$ , then we have found our answer. If  $a[mid] < 13$ , then we know that the target index is more than the mid. So, we drop the lower half by replacing left with mid. If  $a[mid] > 13$ , then we know that the target index is less than the mid. So, we will replace right with mid. We will continue this process till we find a match or till  $left = right$ .

In this method, the size of the array to be searched gets halved in each iteration. So, the worst case number of the required iterations is  $\log_2 n$ . We are not interested in the base of the logarithm. So, we write that the time complexity =  $O(\log n)$ . This is better than  $O(n)$ .

Another example is the Fourier transform. The time complexity of the discrete Fourier Transform is  $O(n^2)$  and that of the fast Fourier Transform is  $O(n \log n)$ .

### 1.1.6 O(2<sup>n</sup>)

Now, consider that we want to find the  $n^{\text{th}}$  Fibonacci number. We can find the  $n^{\text{th}}$  Fibonacci number using the recurrence relation:

$$Fibo(n) = Fibo(n-2) + Fibo(n-1) \text{ and } Fibo(0) = Fibo(1) = 1.$$

Let us write a recursive Python function for it:

```

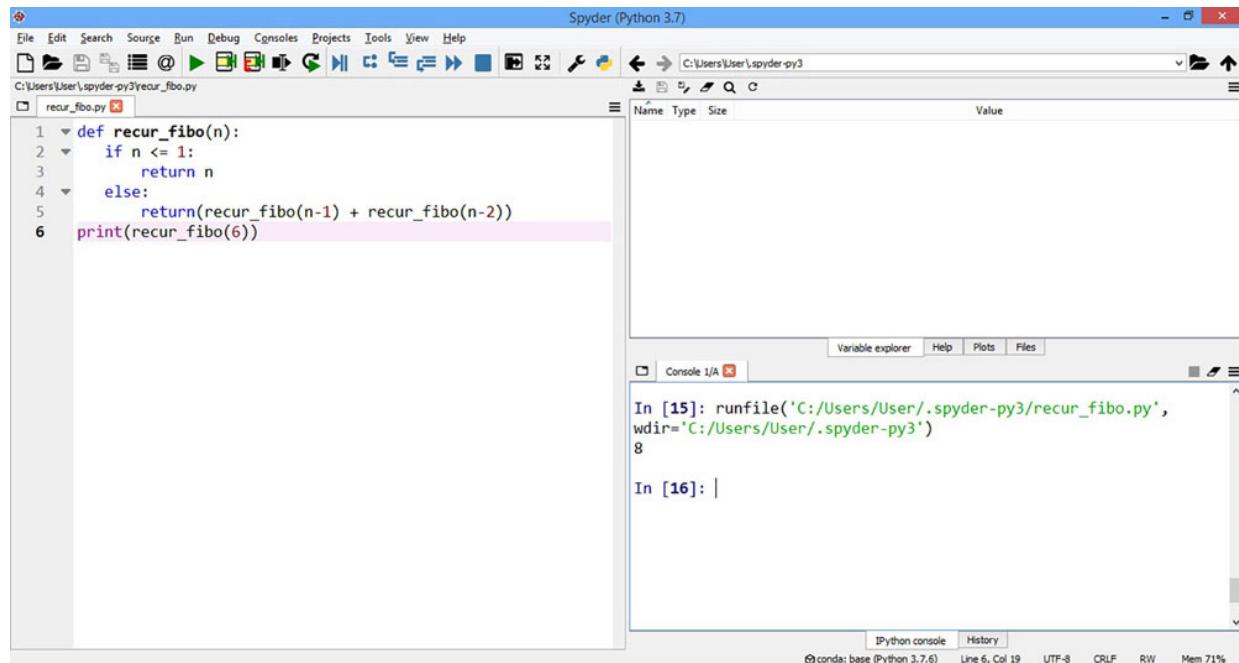
1. def recur_fibo(n):
2.     if n <= 1:
3.         return n
4.     else:
5.         return(recur_fibo(n-1) + recur_fibo(n-2))
6. recur_fibo(6)

```

## Output:

6

Refer to the following screenshot:



**Figure 1.4:** Screenshot for the example of Fibonacci numbers

Here, we observe that each iteration is spawning two more iterations. Thus, if we increase  $n$  by 1, it will double the number of iterations. Therefore, the time complexity is  $O(2^n)$ .

### 1.1.7 Space complexity

The concept of space complexity is similar to that of time complexity. Here, we find how much extra memory space is required by the algorithm as a function of  $n$ . It is also expressed in the big O notation.

If the algorithm does not require any extra space or requires a constant space irrespective of the size, then, the space complexity will be  $O(1)$ .

If an algorithm allocates extra space proportional to  $n$ , then, its space complexity is  $O(n)$ .

## **1.2 Linear Data Structures in Python**

The data structure is a way of organizing data in the computer memory. We choose a data structure depending on the efficiency of intended operations on the given data. The most common operations are insertion, deletion, updating, sorting, and searching. If we can organize the data in a sequential manner such that each element has a predecessor and a successor, and the collection has a first and a last element, it is called a linear data structure. They can be traversed in a single run. The simplest linear structure is a 1-dimensional array. A 2-D array, that is, a matrix is also a linear data structure, though it has two dimensions because an element  $(m, n)$  is mapped to a single dimension array element  $(mN + n)$ , assuming the row and the column numbering starts from 0. Other examples of linear data structures are dynamic array (where the size grows according to the requirement), queue, deque (Double-ended queue), stack, and linked list. The examples of non-linear data structures are hash tables, trees, and graphs. In Python, the dynamic array, queue, and stack are implemented by a single class, named list. The class deque is available in a module, named collections. The string is a special case of the dynamic array where the data elements are ASCII characters and a special set of operations are defined for them. We will study them in detail in the coming sections.

### **1.2.1 The list class**

Python has a large set of useful functions for lists. The most frequently used functions are described here. For the complete list, the reader may refer to the Python documentation.

An object of the list class can be created by enclosing a set of elements in the square brackets. For example:

A = [3, 5, 7]

A structure similar to the list is a tuple. It can be created by enclosing the items in round brackets instead of the square brackets. For example:

```
t = (3,4,5,6)
```

A tuple is immutable, that is, it cannot be changed after it is declared. We cannot add or delete its elements or alter its values. We can only access individual elements just like the list elements. For example:

t[0] means integer 3

a = t[1:] means a tuple a = (4,5,6)

The elements of a list need not be of the same type. For example:

```
a=[1,'e',(1,2,3),[4,5,6]]
```

The elements of a tuple also need not be of the same type. For example:

```
t=(1,'e',(1,2,3),[4,5,6])
```

If we enclose the elements in braces, it creates a set that is a non-linear data structure. For example:

```
s = {1,2,3}
```

We will study sets in the next chapter.

The list is basically an array. The individual elements of a list can be accessed by specifying the index in the square brackets. For example:

```
a = [1,2,3,4,5]
```

a[0] means 1, a[1] means 2, and so on. Indices can be also negative. a[-n] means  $n^{th}$  element from the end. a[-1] means the last element.

A subset of the array can be obtained by specifying the range of indices. For example:

b= a[1:3] assigns value [2,3] to b. Note that the element at the end of index 3 is not included.

The Length of an array is obtained by the “len” function.

```
print(len(a)) prints 5.
```

The indices of a are in the range 0 to len(a) – 1.

If we give a command a[5] = 8, we get an error because there is no element with index 5. For adding elements, we must use the “append” function. For example:

```
a.append(8)
```

This is equivalent to pushing an element on a stack. The inverse operation `pop` is also available.

`b = a.pop()` removes the last element in the array (8) and assigns it to `b`. Both “append” and “pop” operations take place with the time complexity of  $O(1)$ .

In general, `pop(n)` removes the  $n^{\text{th}}$  element from the array. The default value of  $n$  is -1.

The `insert( $n$ , element)` function inserts the element at index  $n$ . The function `insert(0, element)` inserts the element at the beginning. The `insert(0, element)` and `pop()` pair can be used to implement a queue. When you insert an element at index 0, all the existing elements have to be pushed by one place. Therefore, the time complexity of the enqueue operation is  $O(n)$ .

## 1.2.2 NumPy arrays

The built-in list class is useful as a dynamic array, queue, or stack. But, if you want to do mathematical operations like matrix multiplication, it is better to use arrays from the NumPy module. The NumPy arrays take less memory space and perform faster mathematical operations.

## 1.2.3 Strings

In Python, the strings are one-dimensional arrays of ASCII characters. Python has a large set of useful functions for strings. The most frequently used functions are described here. For the complete list, the reader may refer to the Python documentation.

A string literal is declared by enclosing the characters in double quotes or single quotes. For example:

```
s = "Hello"
```

```
s = 'Hello'
```

But basically, it is a list, so it can also be expressed as:

```
s = ['H','e','l','l','o']
```

The individual elements can be accessed just like list objects. For example:

```
s[1] means 'e'
```

`s[2:]` means ‘llo’

The function `len(s)` return returns the length of the string `s`.

The strings can be concatenated by the ‘+’ operator. Thus ‘abc’+’123’ gives ‘abc123’.

The string class has a useful function “`zfill(n)`” which pads zeroes to the left of a string to make its length equal to `n`. For example:

`a = "abc"`

`a.zfill(5)` gives “00abc”.

We will use this function in question 8.

## 1.3 Sorting and Searching

Sorting and searching are the two most widely used operations on data in practice. Sorting means rearranging the elements of an array in ascending or descending order. Searching means finding a data item in a collection. A lot of different sorting algorithms have been developed. Each has its own merits and demerits with respect to the nature of the data. We have to choose an algorithm that suits our particular data distribution. A comparison of the complexities of various algorithms is as follows:

Algorithm	Best case	Average case	Worst case
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$

## Searching

The purpose of searching can be one of the following:

1. We want to check if the given data item exists in a collection.
2. The data consists of a collection of key:value pairs. We want to retrieve the value corresponding to a given key.

If the collection is in the form of a haphazard unsorted array, in that case, the only way of searching an item is by scanning each element of the array and comparing it with the search target. This is called **linear search**. The number of comparisons in the worst case will be  $n$ , where  $n$  is the size of the array. In the best case, we will hit the target in the first comparison, and in the average case, we will need  $n/2$  comparisons. The worst case time complexity is thus  $O(n)$ .

If the data is arranged in a sorted order, we can adopt a smarter searching strategy called the **binary search**, which achieves the time complexity of  $O(\log n)$ . In this strategy, we keep on halving the range of the array to be searched in each iteration till we converge on the item of interest. Questions 1, 7, 10, and 14 are based on binary search. These examples will teach you to tweak the standard binary search algorithm to the given requirements. We can solve question 2 also with the help of the binary search technique. Readers are encouraged to try it. A detailed explanation of the technique is given in question 1.

It is also possible to store the data in a “binary search tree” instead of an array. We can perform a search in the binary search tree with the complexity of  $O(\log(n))$ . It is discussed in [chapter 5](#).

Another possible data structure is a “set” which is based on the Hash tables. It is explained in [chapter 3](#).

## [1.4 Question 1: What is the position of a target in a given sorted array?](#)

Most problems require a two-pointer strategy. Sometimes, both the pointers move in the same direction with the same speed, which results in a sliding window strategy. Sometimes, one pointer moves faster, which results in an expanding window strategy. Sometimes, the pointers move in opposite directions, which results in a pinching window strategy. This is what we will use in the present problem and learn the binary search technique.

## Problem statement

You are given a sorted array and a target value. Search the target value and return its index if it is found. If not, then return the index to where it should be inserted in order. You may assume that there are no duplicates in the array.

Let us illustrate this with the help of some examples.

In all the examples, the array is [1,3,5,6,9].

**Example 1:** Target is 5

**Output:** 2.

**Explanation:** Target 5 appears at position 2 in the array.

**Example 2:** Target is 2.

**Output:** 1

**Explanation:** Target 2 does not appear in the array. It will be inserted at position 2.

**Example 3:** Target is 7.

**Output:** 4

**Explanation:** Target 7 does not appear in the array. It will be inserted at position 4.

**Example 4:** Target is 0.

**Output:** 0

**Explanation:** Target 0 does not appear in the array. It will be inserted at position 0.

## Solution format

Your program may be tested by an automatic evaluation platform. Each platform dictates a standard format compatible with the test environment. It defines the class name, function name, argument sequence, and type and return value type. The format may be different for different platforms. In this book, we will follow the given format. In addition, you also need to write a driver code to test your code before submitting:

1. **class** Solution:

```
2.     def searchInsert(self, nums: List[int], target: int) ->  
int:
```

**Note:** I have already stated in the foreword that you should be familiar with the basic Python before trying these problems. However, I will explain some points where Python nubies are likely to stumble.

Two points you may not be familiar with at the basic level are as follows:

- Function call syntax - `nums: List[int]` and `->int`. They are nothing but typing hints that indicate the data type of the arguments and the return value. Python is a dynamically typed language, so we don't have to declare the data types of the variables. But typing hints improve program readability and reduce the chances of errors. They are ignored by the interpreter. They are available from Python 3.5 onwards. These notations are defined in the `typing` module. For using them, you need to insert the following line at the beginning of the code:

```
from typing import List
```

`nums: List[int]` denotes that the parameter `nums` is a list type and `->int` denotes that the returned value is `int` type.

- If you have not studied the object-oriented programming in Python, you might also be wondering about `self`, which is the first parameter of the `searchInsert` function. Since the `searchInsert` is a member function of the class `Solution`, its first parameter has to be `self`. In C++ and Java, it is implicitly assumed, but in Python, it must be explicitly written. Otherwise, it is omitted when you make a call to the function. The driver code requires that you instantiate an object of `Solution` class and call its method `searchInsert`.

## Strategy

We will study two approaches, namely, linear search and binary search. The purpose is to get an idea about various strategies and their relative advantages and disadvantages.

## Approach 1 – Linear Search

We will traverse the array from the lowest value with the iterating variable  $i$ . When we find a number equal to the target, we return the index  $i$ . If we reach a greater number, it means that the target is not present in the array. Even in this case, we will return the index  $i$  because this is the place where it would be rightfully inserted.

## Python code

The following code implements this approach:

```
1. from typing import List
2. class Solution:
3.     def searchInsert(self, nums: List[int], target: int) ->
int:
4.         if len(nums)==0: return 0
5.         for i in range(len(nums)):
6.             if nums[i]>=target:
7.                 return i
8.         return len(nums)
9. #Driver code
10. sol=Solution()
11. print(sol.searchInsert([1,3,5,6,9], 5))
12. print(sol.searchInsert([1,3,5,6,9], 2))
13. print(sol.searchInsert([1,3,5,6,9], 7))
14. print(sol.searchInsert([1,3,5,6,9], 0))
```

## **Output:**

```
2
1
4
0
```

Refer to the following screenshot:

The screenshot shows the Spyder Python IDE interface. On the left, the code editor displays two classes: `Solution` for linear search and `Binary search`. The `Binary search` class contains a `searchInsert` method that performs a binary search on a sorted list of integers to find the index where a target value should be inserted. The code includes comments explaining the logic of adjusting the `low` and `high` pointers based on the comparison between the target and the middle element. On the right, the variable explorer shows a single variable `sol` of type `Solution` with size 1 and value "Solution object of `_main_` module". Below the variable explorer is the IPython console window, which shows the output of running the script. The command `In [18]: runfile('G:/leet_python/399_1_code/search_insert_pos.py', wdir='G:/leet_python/399_1_code')` was run, followed by the output values 2, 1, 4, and 0, each on a new line. The console window also shows the prompt `In [19]: |`.

**Figure 1.5:** Screenshot for the example of Fibonacci numbers

## Complexity Analysis

The time and space complexity analysis is presented here.

### Time complexity

We have one loop in the program. Whenever we run a loop  $n$  times and the execution time of each iteration is constant, say  $k$ , then the total time will be  $nk$ . As explained in *Section 2.1.1*, the time complexity is  $O(n)$ .

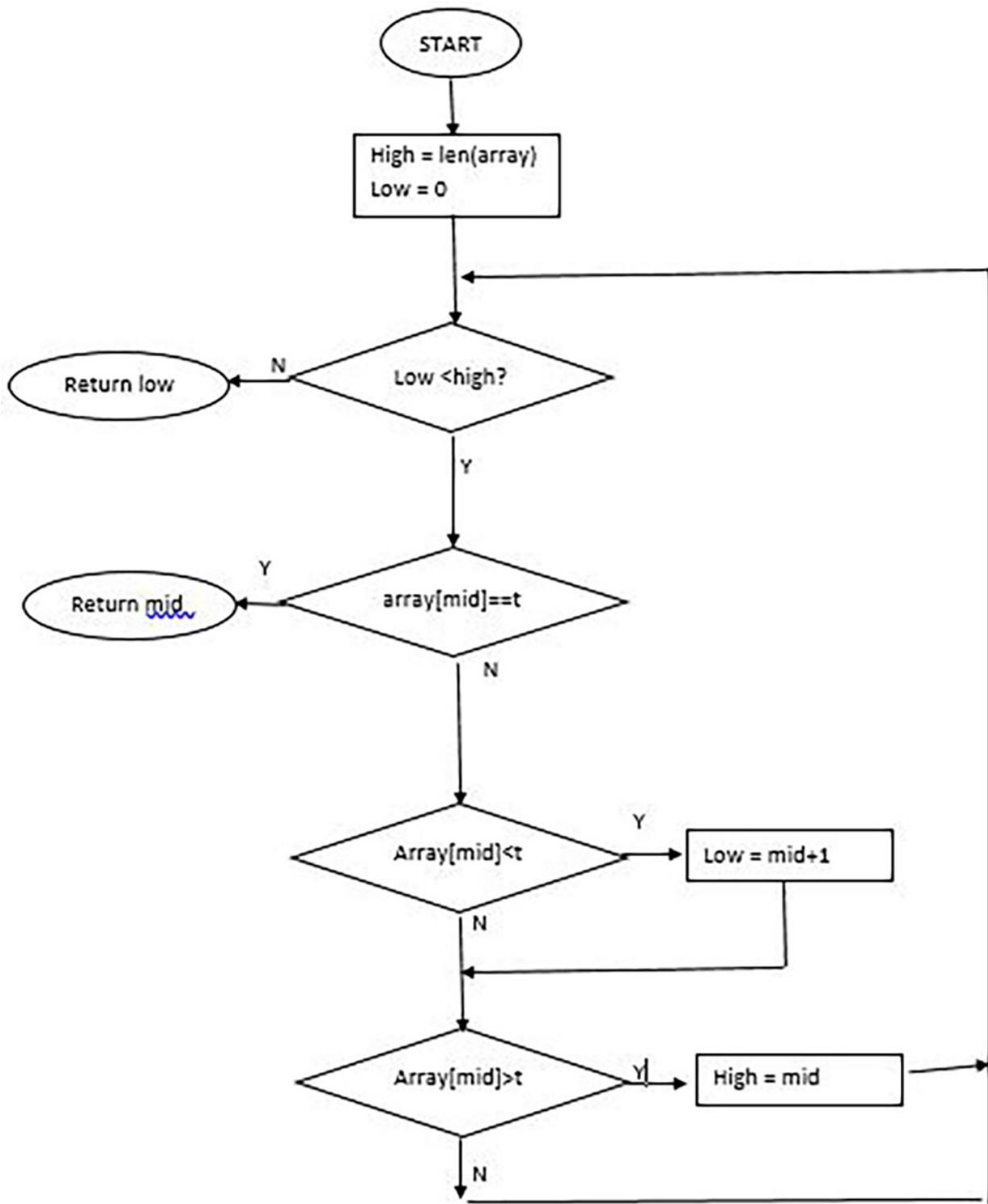
### Space complexity

The function arguments need memory for storage. But for finding the space complexity, we do not count that. We have to count only the extra space required to hold the temporary data. In this case, we are using only one extra variable,  $i$ . The memory requirement is constant, that is, independent of  $n$ . Therefore, the space complexity is  $O(1)$ .

## Approach 2 – Binary Search

This algorithm is called the binary search because in each iteration we will divide the search window into two halves and find out in which half our target lies. The low and high indices define the extent of the window. If the element at the midpoint is equal to the target, we have found the answer. If it

is less than the target, the answer lies in the upper half, so we will discard the lower half by making  $\text{low} = \text{mid} + 1$ . If it is greater than the target, the answer lies in the lower half, so we will discard the upper half by making  $\text{high} = \text{mid}$ . We will repeat this process as long as  $\text{low}$  is less than  $\text{high}$ . If the program terminates with  $\text{low} = \text{high}$ , the element does not exist and the right place for insertion is the lower index. The program flowchart is shown in the following figure:



**Figure 1.6:** Flowchart

This problem is slightly different from a standard binary search problem where we simply find the index of the target, or return an error when it is not

found. Here, we need to find a proper place to insert, rather than return an error. Consider that the target is greater than the highest element and the high is initialized to the index of the highest element. In the end, the mid reaches the index of the highest element and the program returns the index of the highest element, which is wrong. So, the high is initialized to  $\text{len}(A)$ , rather than  $\text{len}(A)-1$ . To reduce the window faster, the standard binary search algorithms sometimes replace the high with  $\text{mid} + 1$ , when array  $[mid]$  is greater than  $t$ . But this program will fail when the array is [1,3] and the target is 2. So, the high is replaced with the mid, rather than  $\text{mid} + 1$ .

Let us illustrate this with an example. Let the array = [1,3,6,7,10,12,15] and the target t=11.

The array elements and their indices are as follows:

Index n	0	1	2	3	4	5	6
Array[n]	1	3	6	7	10	12	15

**Step 0:** low = 0 and high = 7.

$$\text{mid} = \text{int}((0+7)/2) = 3$$

As array[3]<11, make low = mid+1 = 4

**Step 1:** low = 4 and high = 7.

$$\text{mid} = \text{int}((4 + 7)/2) = 5$$

As array[5]>11, make high = mid = 5

**Step 2:** low = 4 and high = 5.

$$\text{mid} = \text{int}((4 + 5)/2) = 4$$

As array[4]<11, make low = mid+1 = 5

**Step 3:** low = 5 and high = 5.

As low is no longer less than high, return 5

So, the element 11 has to be inserted at index 5.

## Python code

The following code implements this approach. The driver code from approach 1 can be used here without any change. So it is repeated here. The

output is also the same. Hence, the screenshot is not attached.

```
class Solution:  
1. class Solution:  
2.     def searchInsert(self, array: List[int], target: int) ->  
int:  
3.         low = 0  
4.         high = len(array)  
5.  
6.         while low < high:  
7.             middle = (high + low) // 2  
8.  
9.             if array[middle] == target:  
10.                 return middle      #We found target  
11.  
12.             if array[middle] < target:  
13.                 low = middle + 1    #Array is smaller than  
target, remove lower half  
14.  
15.             if array[middle] > target:  
16.                 high = middle    #Array is bigger than target,  
remove upper half  
17.         return low           #Come here when low >= high
```

## Complexity Analysis

The time and space complexity analysis is given below.

### **Time complexity**

Since we are running a binary search, the time complexity is  $O(\log n)$  as explained in *Section 1.1.5*.

### **Space complexity**

Since we are not using any size-dependent extra space, the space complexity is  $O(1)$ .

## 1.5 Question 2 – Is the given integer a valid square?

This simple problem explains how an algebraic identity sometimes reduces the time complexity of a solution.

## **Problem statement**

Write a function that returns `True` if a positive integer supplied to it is the square of some integer, otherwise returns `False`. However, you are not allowed to use any library function like `sqrt`.

Let us illustrate with the help of two examples.

### **Example 1:**

**Input:** The given number is 25.

**Output:** True

**Explanation:** 25 is square of 5.

### **Example 2:**

**Input:** The given number is 14.

**Output:** False

**Explanation:** We cannot find an integer whose square is 14.

## **Solution format**

Your function must be in the following format:

```
class Solution:  
    def isPerfectSquare(self, num: int) -> bool:
```

## **Strategy**

First, we will study the linear search approach, which is simple to think and implement, but inefficient. Then, we will present an ingenious approach based on an algebraic identity.

## **Approach 1: Linear Search**

We will iterate for  $i = 1$  to  $i = num$ , and if we find that  $i*i = num$ , we will return true. The range is defined as  $(1, num+1)$ . If you define it as  $(1, num)$ , the function misbehaves for  $num = 2$ .

## Python code

The following code implements this approach. We will write a driver code that tests all the numbers in the range of 1 to 9:

```
1. class Solution:  
2.     def isPerfectSquare(self, num: int) -> bool:  
3.         for i in range(1,num+1):  
4.             if num == i*i: return True  
5.         return False  
6. #Driver code  
7. sol=Solution()  
8. for i in range(1,10):  
9.     print(i,sol.isPerfectSquare(i))
```

## **Output:**

```
1 True  
2 False  
3 False  
4 True  
5 False  
6 False  
7 False  
8 False  
9 True
```

Refer to the following screenshot:

The screenshot shows the Spyder Python IDE interface. The left pane displays the code for two classes: `Linear search` and `Arithmetic progression`, both implementing the `isPerfectSquare` method. The right pane shows the Variable explorer with variables `i` (int, 9) and `sol` (Solution object of `__main__` module). The IPython console at the bottom shows the output of the driver code, which prints the result of `sol.isPerfectSquare(i)` for each integer from 1 to 9.

```

1
2 class Solution:#Linear search
3     def isPerfectSquare(self, num: int) -> bool:
4         if num==1:return True
5         for i in range(1,num//2 + 1):
6             if num == i*i: return True
7         return False
8 ...
9 class Solution:#Arithmetic progression
10    def isPerfectSquare(self, num: int) -> bool:
11        sum = 0;i=1
12        while sum < num:
13            sum += i
14            if sum == num: return True
15            i += 2
16        return False
17 ...
18 #Driver code
19 sol=Solution()
20 for i in range(1,10):
21     print(i,sol.isPerfectSquare(i))

```

Name	Type	Size	Value
<code>i</code>	int	1	9
<code>sol</code>	Solution	1	Solution object of <code>__main__</code> module

```

wdir='G:/leet_python/399_2_code'
1 True
2 False
3 False
4 True
5 False
6 False
7 False
8 False
9 True

```

**Figure 1.7:** Screenshot for the example of valid squares

## Complexity Analysis

The time and space complexity analysis are as follows.

### Time complexity

Due to a single loop, the time complexity is  $O(n)$ .

### Space complexity

Since we are not using any extra space, the space complexity is  $O(1)$ .

## Approach 2 – Summation of Arithmetic Progression

Let us recall that the sum of  $n$  odd numbers is  $n^2$ . Check:

$$1 + 3 = 4$$

$$1 + 3 + 5 = 9$$

$$1 + 3 + 5 + 7 = 16$$

$$1 + 3 + 5 + 7 + 9 = 25$$

We will create a variable  $i$  to represent the odd numbers. It will start at 1 and get incremented by 2 in each iteration. We will go on adding odd numbers to

the variable named “sum”. After addition, if the *sum* equals the number *num*, we exit the loop and return true. If the *sum* exceeds *num*, we return false.

## Python Code

The following code implements this approach. The driver code can be shared with the last problem. As the output is the same, the screenshot is not given:

```
1. class Solution:  
2.     def isPerfectSquare(self, num: int) -> bool:  
3.         sum = 0;i=1  
4.         while sum < num:  
5.             sum += i  
6.             if sum == num: return True  
7.             i += 2  
8.         return False
```

## Complexity Analysis

The time and space complexity analysis are given below.

### **Time complexity**

As  $n$  terms summed gives  $n^2$ , the time complexity is  $O(\sqrt{n})$ .

### **Space complexity**

Since we are not using any extra space, the space complexity is  $O(1)$ .

## 1.6 Question 3 – How will you move zeroes in a given array to the end?

In this problem, we will hone our skills in the basic array manipulations and learn to employ a two-pointer sliding window technique.

### Problem statement

The array called *nums* contains some integers, including zeroes. Write a program to pick all the 0's and move them to the end of the array. Slide the non-zero elements to fill the gap while maintaining their relative order.

Further restrictions:

1. You should not create another array for the output. You must do this in place.
2. Minimize the total number of operations.

Let us illustrate this with an example.

**Example:**

**Input:** The array is [0,1,0,3,12]

**Output:** [1,3,12,0,0]

**Explanation:** There are zeroes in the input array at positions 0 and 2. Move them to the end and remove the gaps by sliding the non-zero numbers to the left.

## Solution format

Your function should be in the following format:

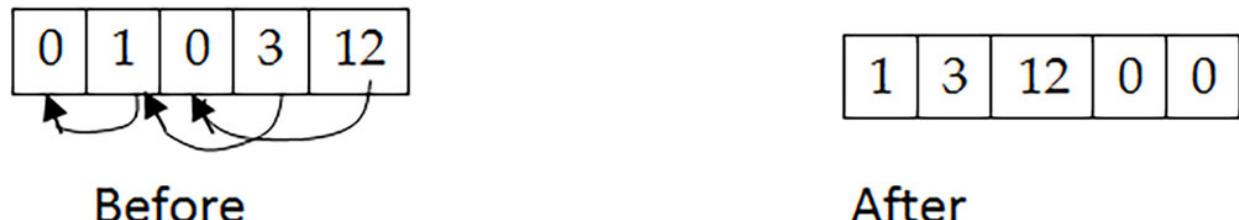
```
class Solution:  
1. class Solution:  
2.     def moveZeroes(self, nums: List[int]) -> None:  
3.         """  
4.             Do not return anything, modify nums in-place instead.  
5.         """
```

Nubies must be wondering about the triple quote marks. It is a way of commenting on a block of the code. It is similar to `/* --- */` in C. In Python version 2, this method was used instead of the type hints. But, the comment is not totally ignored by the interpreter. Block comment is called a **Docstring**. We can give a name to it and access it in a program. We will see its practical use in *Section 1.10.3*.

## Strategy

The simplest thing that comes to mind is to scan the array and whenever a 0 element is encountered, shift all the further elements down by one place and fill the last element with 0. But obviously, such an approach will be very wasteful of time. A better approach would be to shift the elements only once. We need to maintain two pointers. One, which points to the element being

examined (e), and the other, to where it will be shifted (s). If the element `nums[e]` is a non-zero, store it in `nums[s]`. Increment both pointers. Continue until the end. Then, fill the remaining elements with zeroes, as shown in the following figure:



*Figure 1.8: Shifting strategy*

## Pseudo code

```
Initialize e and s with 0.  
For e from 0 to len(nums)-1  
If nums[e] is not zero,  
shift it to nums[s] and increment e and s.  
Else  
    Increment e  
While s < len(nums)  
    Num[s] = 0
```

## Python code

The following code implements this approach:

```
1. from typing import List  
2. class Solution:  
3.     def moveZeroes(self, nums: List[int]) -> None:  
4.         """  
5.             Do not return anything, modify nums in-place instead.  
6.         """  
7.         e=0  
8.         s=0  
9.         for e in range(0, len(nums)): # shift non zero  
elements
```

```

10.         if nums[e]:
11.             nums[s] = nums[e]
12.             s += 1
13.             e += 1
14.         else:
15.             e+=1
16.     while s < len(nums): # fill remaining array with
zeroes.
17.         nums[s]=0
18.         s += 1
19. #Driver code
20. nums = [0,1,0,3,12]
21. sol = Solution()
22. print(nums)
23. sol.moveZeroes(nums)
24. print(nums)

```

### **Output:**

```
[0, 1, 0, 3, 12]
[1, 3, 12, 0, 0]
```

### **Complexity Analysis**

The complexity analysis is as follows:

- Time complexity: Since we run the loop  $n$  times, the time complexity is  $O(n)$ .
- Space complexity: Since we don't require any extra space, the space complexity is  $O(1)$ .

### **1.7 Question 4 – How many boats are required to save people?**

Many times, we have to allocate resources to task subject to some constraints. This problem illustrates the thinking process for such problems.

## Problem statement

We are given an array called `people` which contain weights of the people waiting to be ferried across a river. The  $i^{\text{th}}$  person has the weight  $\text{people}[i]$ . Each boat is designed to carry the weight up to a certain limit.

Each boat can carry at the most two people at a time, provided that the sum of the weights of those people does not exceed the given limit.

Return the minimum number of the boats to be required to carry all the people.

Let us illustrate this with the help of a few examples.

### **Example 1:**

**Input:** The `people` array is [1,2] and the limit is 3

**Output:** 1

**Explanation:** 1 boat carries people (1, 2), as the limit is not exceeded.

### **Example 2:**

**Input:** The `people` array is [3,2,2,1] and the limit is 3

**Output:** 3

**Explanation:** We need 3 boats with people (1, 2), (2), and (3).

**Example 3:** The `people` array is [3,5,3,4] and the limit is 5

**Output:** 4

**Explanation:** We need 4 boats with people (3), (3), (4), (5).

**Note: You may assume the following constraints on the data.**

- $1 \leq \text{people.length} \leq 50000$
- $1 \leq \text{people}[i] \leq \text{limit} \leq 30000$

## Solution format

Your solution should be in the following format:

```
1. class Solution(object):
2.     def numRescueBoats(self, people, limit):
```

## Strategy

We will start with the allocation loop. We will take a new boat and allot it to the heaviest person. If there is room, we will accommodate the lightest person in the same boat, otherwise, the heavier person has the boat for himself. We will continue like this until all the people are carried.

In order to quickly identify the heaviest and the lightest person among the people waiting, we will sort the array in ascending order using the Python library function `sort`. We will set up two pointers, one at each end. We will apply a *pinching window* approach. As the array is in ascending order, the pointer *low* is for the lighter end, and the pointer *high* is for the heavier end. When a heavy person is dispatched, *high* is decremented, and when a light person is dispatched, *low* is incremented. The loop ends when these pointers meet.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def numRescueBoats(self, people, limit):  
3.         people.sort() # Arrange weights in ascending order  
4.         print(people)  
5.         low=0 ;#Set up a pointer at low end  
6.         high=len(people) - 1 # Set up pointer at high end  
7.         n_boats = 0 #Initialize boats counter to 0  
8.         # We pinch the array between advancing low & reducing  
high  
9.         #When low reaches high, we finish  
10.        while low <= high:  
11.            n_boats += 1 # Take a new boat  
12.            #Heavy person takes the boat, so always  
decrement high  
13.            #If boat can take accomodate a light person,  
increment low  
14.            if people[low] + people[high] <= limit:  
15.                low += 1  
16.                high -= 1
```

```
17.         return n_boats
18. people = [3,2,2,1]
19. limit = 3
20. sol = Solution()
21. print(sol.numRescueBoats(people, limit))
```

## Output:

```
[1, 2, 2, 3]
3
```

## Complexity Analysis

The complexity analysis is as follows.

### Time complexity:

Python internally uses the Timsort algorithm for sorting, which has the time complexity of  $O(n \log n)$ . After sorting, we run the allotment loop  $n$  times. Thus, the time complexity is  $O(n \log n + n)$ .

In the big O notation, we take only the major term. So, the *time complexity* =  $O(n \log n)$

### Space complexity:

In the worst case, Timsort requires  $n/2$  pointers. The allotment process does not require any extra space. So, the space complexity is  $O(n)$ .

## 1.8 Question 5 – Is the given array a valid mountain array?

In many situations you need to detect certain patterns in the given data, e.g., continuously rising, continuously falling, rising and falling, falling and rising, etc. This problem explains the strategy to handle them.

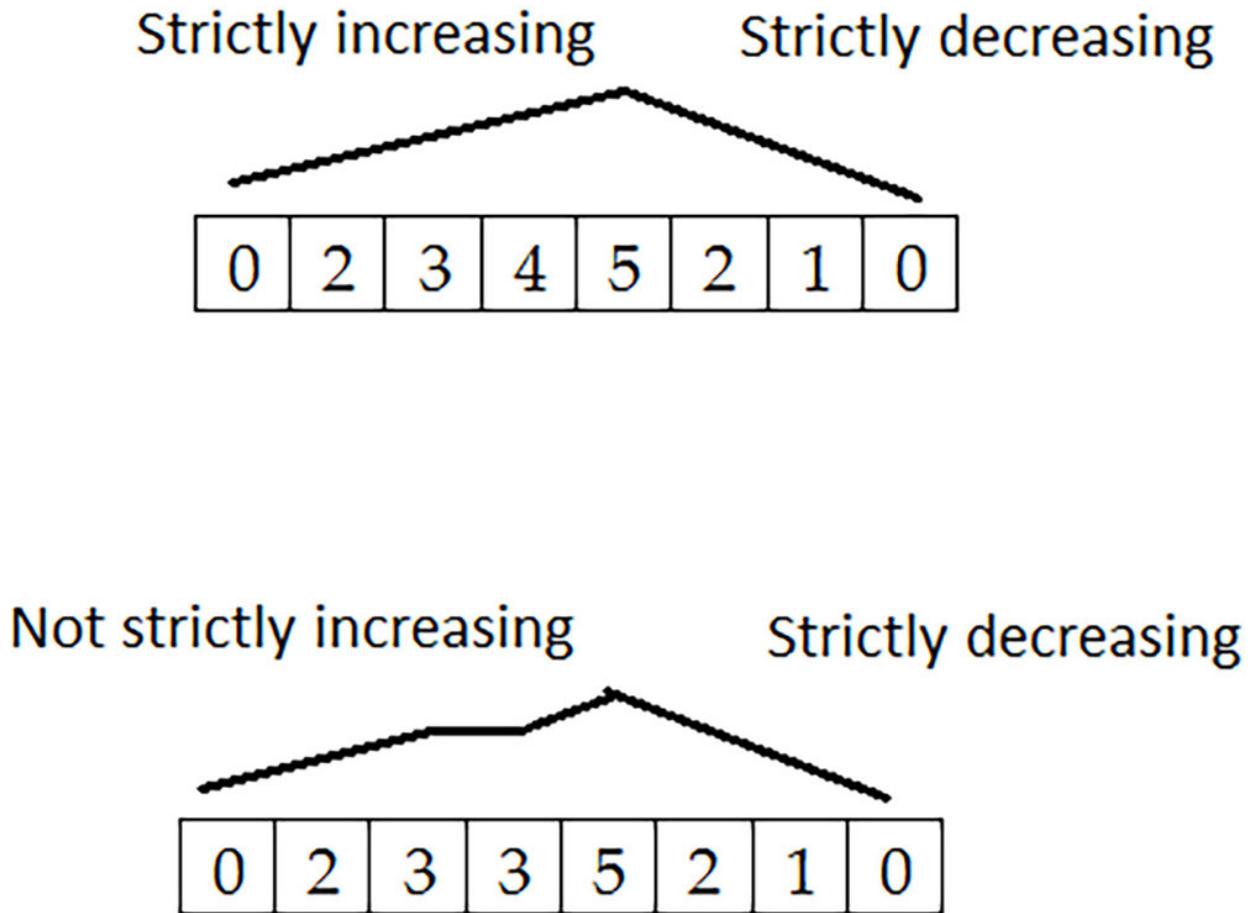
### Problem statement

You are given an array A of integers. Write a function that returns *true* if and only if it is a valid mountain array, that is, its values rise to a peak and then fall. But rise and fall should be strictly monotonic.

We could mathematically say that A is a mountain if:

- $A.length \geq 3$
- There exists some  $i$  with  $0 < i < A.length - 1$  such that:
  - $A[0] < A[1] < \dots < A[i-1] < A[i]$
  - $A[i] > A[i+1] > \dots > A[A.length - 1]$

The following figure shows examples of some valid and invalid mountain arrays:



*Figure 1.9: Mountain array examples*

**Note:**

- $0 \leq A.length \leq 10000$
- $0 \leq A[i] \leq 10000$

Let us illustrate this with the help of some examples.

### **Example 1:**

**Input:** Array is [3,1]

**Output:** False

**Explanation:** There is no increasing part.

### **Example 2:**

**Input:** Array is [2,7,7]

**Output:** False

**Explanation:** 7,7 is not strictly increasing.

### **Example 3:**

**Input:** [0,4,2,1]

**Output:** True

**Explanation:** It has both the increasing and decreasing parts.

## **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def validMountainArray(self, A: List[int]) -> bool:
```

## **Strategy**

In simple words, the array is valid if it is at least three elements long, starts with the strictly ascending elements till it reaches a peak, and then has strictly descending elements till it ends. We will scan the array with the iterator  $i$ . We will create a flag named `ascending` to indicate that we are in an ascending state. In this state, the elements must be strictly increasing. Once we find a descending element, we will make the flag `ascending` false and look for the descending values. Remember to ensure at the end that both the ascending and the descending parts are present.

## **Python code**

The following code implements this strategy:

```

1. from typing import List #Not needed when submitting to eval
platform
2. class Solution:
3.     def validMountainArray(self, A: List[int]) -> bool:
4.         if len(A) < 3: return False # A must have at least 3
elements
5.         ascending = True#Set ascending state
6.         for i in range(len(A)-1):
7.             if A[i] == A[i+1]: return False #Two consecutive
elements must not be equal
8.             if ascending:
9.                 if A[i] > A[i+1]: ascending = False ##
Descending state set
10.            else:
11.                if A[i] <= A[i+1]: return False #Check
descending condition
12.            #Ensure that ascending and descending sequences
exist
13.            if A[-2] > A[-1] and A[0] < A[1]:#Ensure that
ascending and desc
14.                return True
15.            else: return False
16. #Driver code
17. nums = [0,1,2,3,4,5,6,7,8,9]
18. sol = Solution()
19. print(nums)
20. print(sol.validMountainArray(nums))

```

## **Output:**

False

## **Complexity Analysis**

The complexity analysis is as follows.

## **Time complexity**

As the loop is executed  $n$  times, the time complexity is  $O(n)$ .

### Space complexity

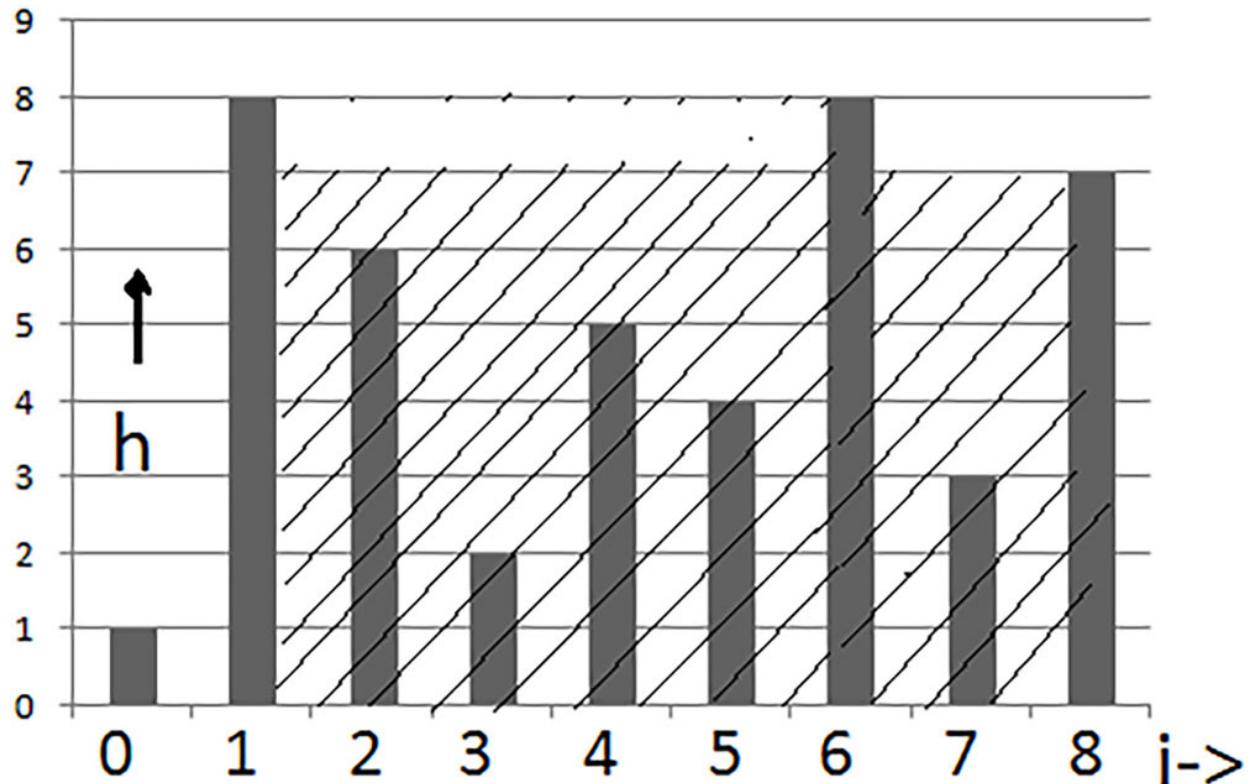
There is no extra space allocation. So, the space complexity is  $O(1)$ .

## 1.9 Question 6 – Which container can store maximum water?

This problem exposes you to an abstract optimization situation. The strategy presented will give you an insight into the required thought process.

### Problem statement

You are given an array  $h$  containing  $n$  non-negative integers  $a_1, a_2, \dots, a_n$ . Each element represents the height of a line in a bar chart. Imagine the lines to be a cross-section of the walls in which the water can be filled. Refer to the following figure:



**Figure 1.10:** Lines for array [1,8,6,2,5,4,8,3,7]

Find two lines, which together with the floor, would form a container, such that the container stores maximum water. The area between the indices  $i$  and  $j$  can be calculated as the difference between the indices multiplied by the minimum heights.

**Note:** You may not slant the container and  $n$  is at least 2.

Let us illustrate the problem with the following example.

**Example:**

**Input:** Let the heights array be [1,8,6,2,5,4,8,3,7].

**Output:** 49

**Explanation:** Consider the left wall at index 1 and the right wall at index 8. The wall on the left is 8 units tall and the wall on the right is 7 units tall. If we fill water in the left wall, it will spill over the right wall. So, the maximum level of water will be 7 units. The base of the container will be the  $8-1 = 7$  units. The hatched section in the figure shows the water. The amount of water is:

$$\begin{aligned} & (j - i) * \min(A[i] - A[j]) \dots(1) \\ &= (8-1) * \min(8,7)=49. \end{aligned}$$

We can try out various combinations of two indices. The maximum area is found to be bound by the indices 1 and 8.

## Strategy

We will consider two approaches, namely, the brute force and the pinching window. The brute force approach is simple to think, but it is inefficient. The pinching window approach is ingenious and efficient.

## Brute force approach

We will consider every possible pair of left and right indices and apply equation (1). We will find which pair maximizes water. Due to nested loops for  $i$  and  $j$ , this algorithm will have the time complexity  $O(n^2)$ . We will not delve into it further and work on a better method.

## Pinching window approach

In a sliding window, we have two pointers that run in the same direction. In a pinching window, we have two pointers, initially at the opposite ends, but both running inwards to pinch the window. Motivation comes from the fact that it is desirable that the base be as large as possible. So, we will start with the widest base, that is,  $left = 0$ , and  $right = \text{len}(A)-1$ . We will create a variable named  $\text{maxwater}$  to keep track of the maximum area. We will initialize  $\text{maxwater}$  with 0. To find if a better solution exists inside the window, we will move the pointer having the smaller height inwards in the hope of finding a greater height inside. If we get a better solution, we will update  $\text{maxwater}$ . The loop will continue as long as the  $left$  is less than the  $right$ .

## Answer format

Your solution should be in the following format.

```
1. class Solution:  
2.     def maxArea(self, h: List[int]) -> int:
```

## Python code

The following code implements this strategy:

```
from typing import List  
  
1. from typing import List  
2. class Solution:  
3.     def maxArea(self, h: List[int]) -> int:  
4.         maxwater = 0 # Initialize maximum variable to 0  
5.         l, r = 0, len(h)-1 # Define initial window of maximum  
size  
6.         while l < r:  
7.             if h[l] <= h[r]:#Find smaller height and shift  
its pointer inwards  
8.                 maxwater = max(maxwater, (r-l) * h[l])  
9.                 l += 1  
10.            else:  
11.                maxwater = max(maxwater, (r-l) * h[r])
```

```
12.             r -= 1
13.         return maxwater
14. #Driver code
15. nums = [1,8,6,2,5,4,8,3,7]
16. sol = Solution()
17. print(nums)
18. print(sol.maxArea(nums))
```

### **Output:**

49

### **Complexity Analysis**

The complexity analysis is as follows.

#### **Time complexity:**

As the loop is executed  $n$  times, the time complexity is  $O(n)$ .

#### **Space complexity:**

There is no extra space allocation. So, the space complexity is  $O(1)$ .

## **1.10 Question 7 – Which was the first faulty version of the software?**

You must be familiar with the algorithm of the binary search in a sorted array. But, this algorithm can be used to model some other practical situations as explained in this problem.

### **Problem statement**

Consider that you manage a software team that produces a package and constantly brings out newer versions. Suddenly, your latest version is found to be faulty by the QC department. But, remember that each version is based on the previous version; so all the versions before the faulty one are also faulty. You need to check all the previous versions to see if they fail under the new QC test.

Suppose you have  $n$  versions  $[1, 2, \dots, n]$ , and you want to find out the first faulty one, which causes all the following ones to be faulty.

The QC department gives you an API of the form `bool isBadVersion(version)` which returns `true` if the version is faulty. Implement a function to find the first faulty version. You shall minimize the number of calls to the API. Let us illustrate this with the help of an example.

### **Example:**

Suppose  $n = 5$ , that is, you have brought out 5 versions from 1 to 5. The QC API returns `False` for versions 3 and less. Some test results are as follows:

```
call isBadVersion(2) -> false
call isBadVersion(3) -> false
call isBadVersion(4) -> true
call isBadVersion(5) -> true
```

Thus, 4 is the first faulty version.

### **Solution format**

Your solution should be in the following format:

1. `class Solution:`
2.   `def firstBadVersion(self, n):`

### **Strategy**

A straightforward approach is a linear search, in which you sequentially call `isBadVersion(i)` for all  $i$  till you get the answer `true`. Its time complexity is  $O(n)$ , and we will not discuss it further.

As you know, the binary search offers the time complexity of  $O(\log n)$ , so, we will use this approach. First, we will check the trivial case of  $n < 2$ . Then, we will set up two pointers, `start`, and `end`. We know that `Version(0)` is good and `version(n)` is bad. Then, as long as the `start` is less than the `end`, we will find the midpoint of `start` and `end` with the instruction `mid = (left+right)//2`. Note that the `//` operator gives the integer division.

If `version(mid-1)` is good and `version(mid)` is bad, we have found our answer. Otherwise, if `version(mid)` is bad, we replace `right` with `mid`, else we replace `left` with `mid`. For testing the program, we need to define a helper

function `isBadVersion(version)`, which returns `true` if `version >= first_bad`.

## Python code

The following code implements this strategy.

```
1. #QC test API: Helper function for testing
2. def isBadVersion(version):
3.     if version >= first_bad:
4.         return True
5.     return False
6. class Solution:
7.     def firstBadVersion(self, n):
8.         left = 1
9.         right = n
10.        while left < right:
11.            mid = (left+right)//2
12.            if isBadVersion(mid):
13.                right = mid
14.            else:
15.                left=mid+1
16.        return left
17. #Driver code
18. sol = Solution()
19. first_bad = 4
20. print(sol.firstBadVersion(8))
```

## **Output:**

4

## Complexity Analysis

The complexity analysis is as follows.

### **Time complexity:**

Since we are using binary search, the time complexity is  $O(\log n)$ .

## **Space complexity:**

There is no extra space allocation. So, the space complexity is  $O(1)$ .

## **1.11 Question 8 – What are all the subsets of a given set of integers?**

Finding all the subsets of a set is not a trivial problem. We will give two approaches to solve it. The first is the systematically adding elements to a null set, and the second is using a binary representation of the  $n$ -bit number.

### **Problem statement**

Consider a set of distinct integers, `nums`. Find all the possible subsets of this set. It is also called the **power set**.

**The solution set must not contain duplicate subsets.**

Let us illustrate this with an example.

#### **Example:**

**Input:** We need to find the subsets of [1,2,3].

**Output:** It is a list of lists as shown below.

```
[  
    [3],  
    [1],  
    [2],  
    [1, 2, 3],  
    [1, 3],  
    [2, 3],  
    [1, 2],  
    []  
]
```

### **Solution format**

Your solution should be in the following format:

1. **class Solution:**

```
2.     def subsets(self, nums: List[int]) -> List[List[int]]:
```

## Strategy

The number of subsets of an  $n$  length array is  $2^n$ . So, the time complexity of the problem is  $2^n$ . We can grow the subsets by systematically adding elements to a null set. Alternatively, we can consider a set of  $n$  bits to represent whether an element is present or absent in a subset. There are  $2^n$  combinations of these bits. We will add the elements corresponding to 1's in the bit pattern.

## Approach 1 – Growing from seeds

The output array will be an array of subsets. The smallest subset will obviously be a null set. Let us use this as a seed and grow bigger subsets by adding elements from `nums`, one by one.

We will begin by initializing the output to `[]`, which represents one member which is a null set.

We will run two nested loops. The outer loop iterates the `nums` array with the iterator `num`. The inner loop iterates the output array with the iterator `curr`.

For each element from `nums`, we will create a `new_output` by appending that element to each subset element of the output, i.e. `curr`. When `num` is added to all the elements of the output, we will append the `new_output` to `output`.

Let us walk through the execution of the preceding example.

Initially:

`nums = [1,2,3]`

`Output = []`

Consider the first element of `nums`, i.e., 1.

`New_array` is initialized to `[]`.

The array, `curr`, takes the only element of the output, i.e., `[]`.

We append 1 to it to get `[1]`. So, the `new_array` becomes `[1]`.

Append it to the output, so, the output becomes `[],[1]`.

Now, consider the second element of `nums`, i.e., 2.

New\_array is initialized to [].

The array, curr, takes values [] and [1] from the output.

We append 2 to both to get [2] and [1,2]. So, the new\_array becomes [[2], [1,2]].

Append it to the output, so, the output becomes [[], [1], [2], [1,2]].

Now, consider the third element of nums, i.e., 3.

New\_array is initialized to [].

The array, curr, takes values [], [1], [2], [1,2] from the output.

We append 3 to all. So, the new\_array becomes [[3], [1,3], [2,3], [1,2,3]].

Append it to the output, so, the output becomes [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]].

## Python code

The following code implements this strategy:

```
1. from typing import List #Not needed when submitting to test
platform
2. class Solution:
3.     def subsets(self, nums: List[int]) -> List[List[int]]:
4.         output = [[]]
5.         for num in nums:
6.             new_output = []
7.             for curr in output:
8.                 new_output += [curr + [num]]
9.             output += new_output
10.            return output
11. #Driver code
12. sol = Solution()
13. print(sol.subsets([1,2,3,4]))
```

## **Output:**

```
[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

## Complexity Analysis

The complexity analysis is as follows.

### **Time complexity:**

The outer loop runs  $n$  times. The inner loop elements double in every loop. So, its time complexity is  $O(2^n)$ . Therefore, the overall time complexity is  $O(n2^n)$ .

### **Space complexity:**

The algorithm produces  $2^n$  subsets of the worst case length,  $n$  each. So, the space complexity is  $O(n2^n)$ .

## Approach 2: Binary Representation

If you see how the subsets are formed earlier, you will realize that a number from  $nums$  is either present or absent. There are  $n$  elements in  $nums$ . So, with each element being given the choice of being present or absent, there can be  $2^n$  combinations. If a bit  $j$  is present in the binary representation, then  $nums[j]$  is present in the subset. Note that LSB of the binary representation corresponds to index 0 in  $nums$ . The combinations are summarized in the following table:

Nums	3	2	1	subset
Index	2	1	0	
Subset 0	0	0	0	[]
Subset 1	0	0	1	[1]
Subset 2	0	1	0	[2]
Subset 3	0	1	1	[1,2]
Subset 4	1	0	0	[3]
Subset 5	1	0	1	[1,3]
Subset 6	1	1	0	[1,2]
Subset 7	1	1	1	[1,2,3]

**Table 1.1:** Binary representation of subsets

In the loop,  $i$  runs from 0 to  $2n - 1$ . At each step, we will find its binary representation. Then, let the index  $j$  run through the bits of the binary representation, and append  $\text{nums}[j]$  to the subset if the bit is 1. Note that the order of finding the bits is LSB first.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def subsets(self, nums: List[int]) -> List[List[int]]:
3.         n = len(nums)
4.         output = []
5.         for i in range(2**n): # Loop over 2^N combinations
6.             temp = i # Convert temp to binary
7.             temparr = [] #Subsets start as empty array
8.             for j in range(n):
9.                 if temp%2: temparr.append(nums[j])
10.                temp = temp//2
11.            output += [temparr] # use square brackets to
make it an array element
12.        return output
13. #Driver code
14. sol = Solution()
15. print(sol.subsets([1,2,3]))
```

## **Output:**

```
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

## Complexity Analysis

The complexity analysis is as follows.

### **Time complexity:**

The outer loop runs  $2^n$  times. The inner loop runs  $n$  times. So, its time complexity is  $O(n)$ . Therefore, the overall time complexity is  $O(n2^n)$ .

### **Space complexity:**

The algorithm produces  $2^n$  subsets of the worst case length,  $n$  each. So, the space complexity is  $O(n2^n)$ .

### We can use some neat Python tricks to simplify the coding

We can use the Python `bin` function to convert an integer into a string of binary digits. E.g.:

```
bin(6) gives "0b110"  
bin(3) gives "0b11"
```

We see two problems in using this function. First, the opening two letters “0b”, and the second, the string is of variable length. We need a fixed length  $n$  for all the strings. The first problem can be solved by using the array slicing.:

```
bin(6).[2:]#gives "110"  
bin(3).[2:]#gives "11"
```

The second problem can be solved by using the `zfill` function of the *string* class. It produces a fixed-length string, filling zeroes to the left if necessary:

```
bin(3).[2:].zfill(3)#gives "011"
```

A list could be created by an internal loop like this:

```
bits = bin(6).[2:]# bits = "110"  
set = [nums[j] for j in range(3) if bits[j] == '1'] # set =  
[1,2]
```

Note that we need to enclose the set in one more pair of square brackets so that it is treated as a list element. Otherwise, it simply adds the individual elements.

## Python code

Let us use these ideas in the following code given. The driver code remains the same as the previous program:

```
1. class Solution:  
2.     def subsets(self, nums: List[int]) -> List[List[int]]:  
3.         n = len(nums)
```

```

4.         output = []
5.         for i in range(2**n): # Loop over 2^N combinations
6.             bits = bin(i)[2:].zfill(n)
7.             output += [[nums[j] for j in range(n) if bits[j]
== '1']]#Double[]
8.         return output

```

### **Output:**

`[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]`

Notice that the sequence of the output is different because we are scanning the binary string from left to right, i.e.,  $j$  is going from MSB to LSB.

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

The outer loop runs  $2^n$  times. The inner loop runs  $n$  times. So, its time complexity is  $O(n)$ . Therefore, the overall time complexity is  $O(n2^n)$ .

### **Space complexity:**

The algorithm produces  $2^n$  subsets of the worst case length,  $n$  each. So, the space complexity is  $O(n2^n)$ .

## **1.12 Background – Measuring the execution time and the time complexity**

We have discussed the time complexity in every problem. But, how do we measure it experimentally? How to compare the execution time of solutions submitted by two contestants? We have postponed this discussion to this point so that you first get a hands-on experience of what you want to measure.

### **1.12.1 Introduction**

The time measurement gets complicated because while your program is running, the OS interrupts it many times to do background tasks. Its effect

on the program is not predictable. One way of measuring the execution time is to submit the task to one of the online evaluation platforms. But you will observe that if the same code is submitted many times, you get a different result and a different percentile. This is because time depends on how much load is currently present on the server.

Time measurement is more accurate if made on a local machine but some amount of randomness is still there. We will study three approaches for measuring the execution time.

1. Using Python function `perf_counter`. It returns the internal time count within seconds.
2. Using Python function `timeit.timeit` for repetitive measurements to get accurate results.
3. Passing parameters to the functions to be tested in order to get a graph of  $n$  versus execution time, that is, the  $O(n)$  graph. We will obtain graphs for some of the problems we discussed earlier.

### 1.12.2 Using `perf_counter`

Let me illustrate this with the Fibonacci example we discussed in *Section 1.1.6* earlier. Python provides a function called `perf_counter()` which returns the current time in seconds. We will call it before executing the Fibonacci function and store the returned value in a variable named `tic`. We will again call it after executing the Fibonacci function and store the returned value in a variable named `toc`. The difference between `tic` and `toc` gives the execution time in seconds. We can multiply it by 1e6 to get the result in microseconds.

The following code illustrates this procedure:

```
1. from time import perf_counter
2. def recur_fibo(n):
3.     if n <= 1:
4.         return n
5.     else:
6.         return(recur_fibo(n-1) + recur_fibo(n-2))
7.
```

```
8. tic = perf_counter()
9. recur_fibo(5)
10. toc = perf_counter()
11. print((toc-tic)*1e6, " Microseconds")
```

When I ran it three times on my computer, the results were as follows:

```
4.664998414227739 Microseconds
9.952000254997984 Microseconds
4.354000338935293 Microseconds
```

Probably 4.35 is the correct time, showing that it was not affected by the background processes. To get good results, we should run the test many times and take the minimum (not average) value of the timings.

### 1.12.3 Use of `timeit`

Python provides just the right function called the `timeit` for our purpose. It is defined in the `timeit` module.

The prototype is `timeit.timeit(stmt, setup, timer, number)`

The arguments are:

- *stmt*. which is the statement you want to measure; it defaults to ‘pass’.
- *Setup*, which is the code you run before running the *stmt*; it defaults to ‘pass’.

We generally use this to import the required modules for our code.

- *Timer*, which is a `timeit.Timer` object. It usually has a sensible default value, that is, `perf_counter`, so you better leave it at default.
- *number* is the number of executions for which *stmt* is run. It has the default value of 100000. You may want to reduce it to get a quicker response.

The return value is the minimum execution time over 100000 trials.

Let us leave the *setup* and the *timer* parameters at their default values but reduce the *number* parameter to 100. Try the following command at the

console to fill a list  $x$  with the numbers from 0 to 9. The command is automatically repeated 100 times:

```
>>>import timeit  
>>>timeit.timeit("mylist = [x for x in range(10)]", number=100)
```

When I ran this 3 times, the results were:

```
8.210600026359316e-05  
8.117299876175821e-05  
8.21069988887757e-05
```

These are much more consistent compared to the direct `perf_count` method. In the above example, `stmt` was a single statement enclosed in quotes. If you want to measure a multiple-line function, you can enclose it into a *docstring* by enclosing it in triple quotes, and giving it a name. For example, consider the following code snippet:

```
1. import timeit  
2. stm='''  
3. N=10  
4. mylist = [x for x in range(N)]  
5. '''  
6. t=timeit.timeit(stm, number=100)  
7. print(t*1e6)
```

### Output:

```
82.4169983388856
```

## 1.12.4 Plotting the time complexity graphs

Now, let us see how we can plot a graph of the execution time vs the size of the input. This will show  $O(n)$  in action. However, the one problem in doing this is that the main namespace is different from the namespace of functions defined in `timeit`. So, we can neither call the functions defined in the main, nor can we access the variables defined in it. The remedy is to use the `setup` parameter of the `timeit` function. In `setup`, we will import the functions

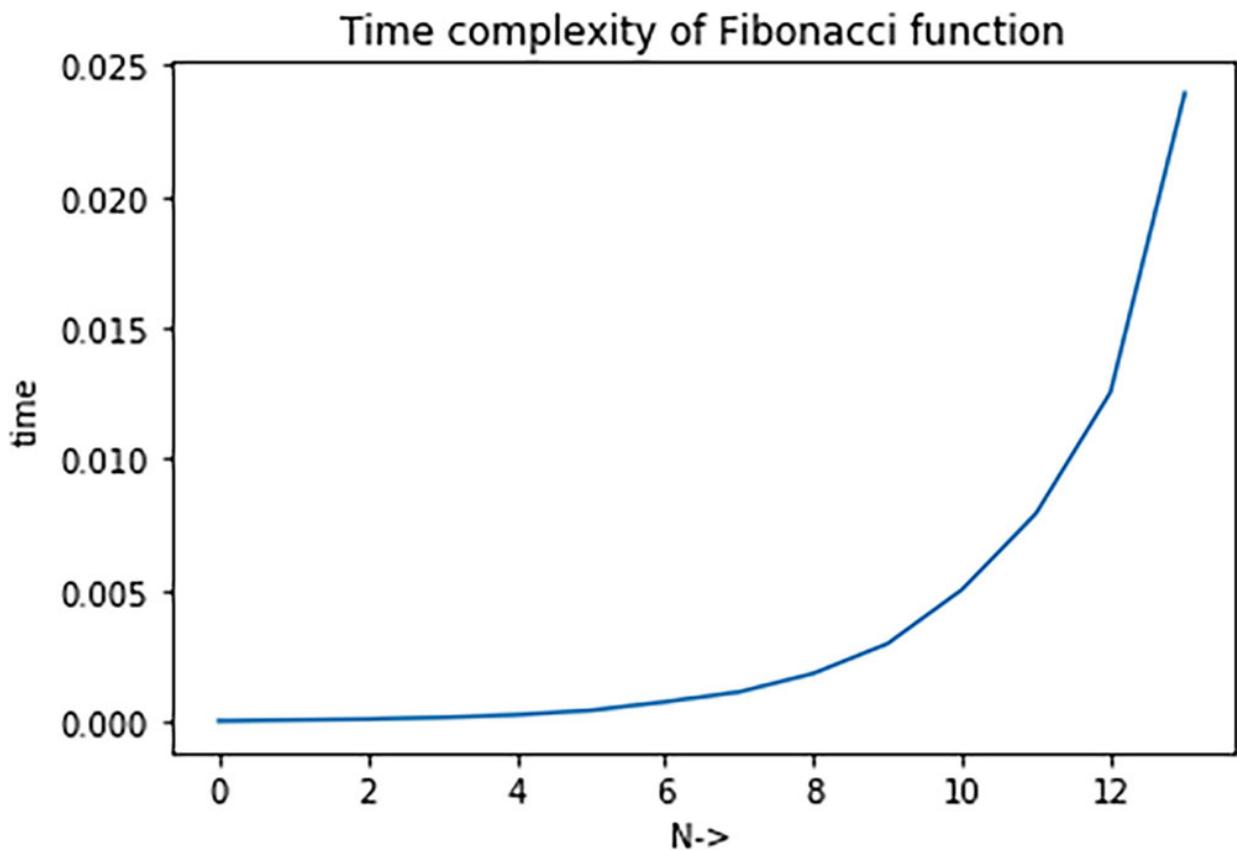
names and variable names defined in the main into `timeit` namespace. Let us illustrate this with the help of the Fibonacci example.

## Python code

The following code implements the procedure:

```
1. import timeit
2. from matplotlib import pyplot as plt
3.
4. def fibo(n):
5.     if n <= 1:
6.         return n
7.     else:
8.         return(fibo(n-1) + fibo(n-2))
9.
10. su=''''
11. from __main__ import fibo,i;           #Import names from
main
12. '''
13. x=[]
14. y=[]
15. for i in range(1,15):
16.     t=timeit.timeit('fibo(i)',setup=su,number=100)
17.     x.append(i)
18.     y.append(t)
19. plt.plot(x,y)
20. plt.xlabel("N->")
21. plt.ylabel("time")
22. plt.title("Time complexity of Fibonacci function")
```

The plot is given in [Figure 1.11](#). This is a typical graph of  $O(2^n)$  complexity.



*Figure 1.11: Time complexity graph for the Fibonacci function*

## Plotting a graph for the sort function

Let us explore the time complexity of the sort function in the Python library.

### Python code

The following code is similar to the code for the Fibonacci example:

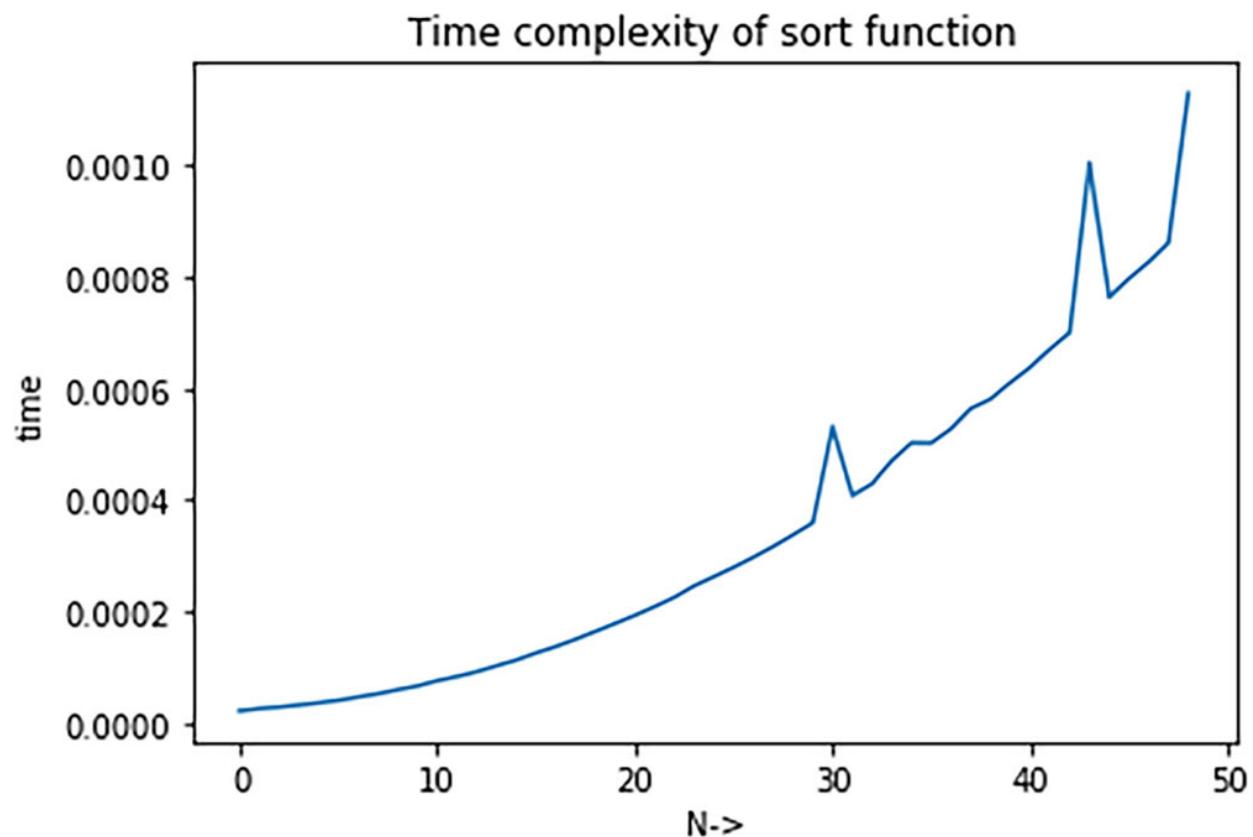
```
1. import timeit
2. import random
3. from matplotlib import pyplot as plt
4. a=[]
5. def sorta():
6.     a.sort()
7.
8. su=''
9. from __main__ import sorta;
```

```

10. '''
11. x=[]
12. y=[]
13. for i in range(1,50):
14.     for j in range(i):
15.         a.append(random.randint(1,100))
16.     t=timeit.timeit('sorta()',setup=su,number=100)
17.     x.append(i)
18.     y.append(t)
19. plt.plot(x,y)
20. plt.xlabel("N->")
21. plt.ylabel("time")
22. plt.title("Time complexity of sort function")

```

Refer to the following figure for the resulting graph:



**Figure 1.12:** Time complexity graph for the Sort function

This is a typical graph of  $O(n \log n)$ . Notice that the graph shows some kinks at N= 30 and N=40. This is because of the vagaries of the background processes or random number generator. If you run the program again, the kinks will appear somewhere else. It is best to ignore them.

## Plotting a graph for the subsets of a set problem

We have already solved this problem in *the earlier section*. Now, let us study its time complexity.]

### Python code

The following code is similar to the previous examples:

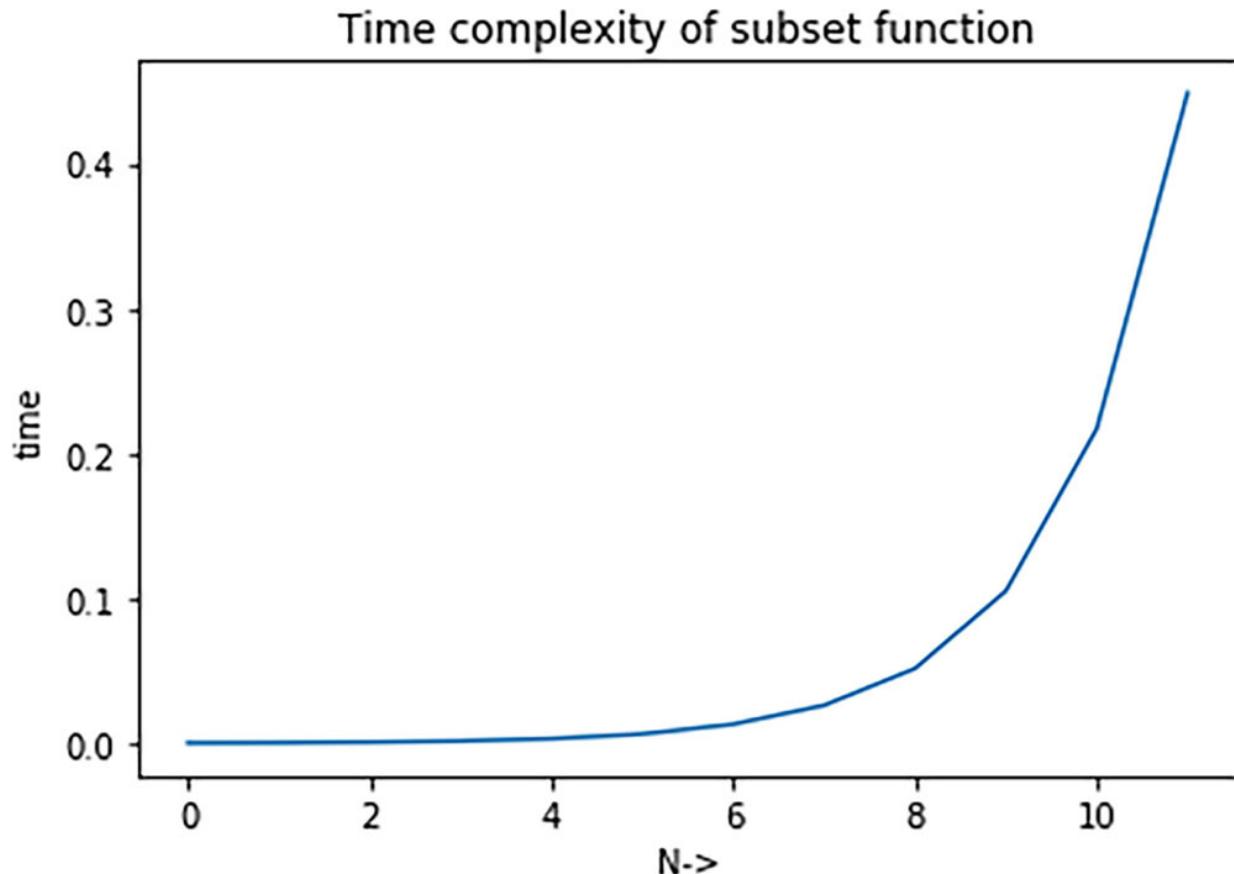
```
1. import timeit
2. import random
3. from matplotlib import pyplot as plt
4. from typing import List
5. a=[]
6.
7. class Solution:
8.     def subsets(self, nums: List[int]) -> List[List[int]]:
9.         output = [[]]
10.        for num in nums:
11.            new_output = []
12.            for curr in output:
13.                new_output += [curr + [num]]
14.            output += new_output
15.        return output
16. sol = Solution()
17.
18. su=''''
19. from __main__ import sol,a
20. '''
21. x=[]
22. y=[ ]
```

```

23. for i in range(3,15):
24.     a=[]
25.     for j in range(i):
26.         a.append(random.randint(1,100))
27.     t=timeit.timeit('sol.subsets(a)', setup=su, number=100)
28.     x.append(i)
29.     y.append(t)
30. plt.plot(x,y)
31. plt.xlabel("N->")
32. plt.ylabel("time")
33. plt.title("Time complexity of subset function")

```

Refer to the following figure for the resulting graph. It exhibits  $O(2^n)$  complexity:



**Figure 1.13:** Time complexity graph for the Subset function

## 1.13 Question 9 – What is the maximum sum of a subarray of a given array?

This problem uses a sliding window approach, and it is a good exercise for manipulating the subsets of an array.

### Problem statement

Given an array of integers and a number  $k$ , find the maximum sum of a subarray of size  $k$ . Note that a subarray consists of a set of elements from the main array in the same sequence. A subset can have its elements in any sequence.

Let us illustrate this with the help of two examples.

**Example 1:** Let the input array be[100, 200, 300, 400], and  $k = 2$ .

**Output:** 700

**Explanation:** Consider the subarrays consisting of a sliding window of size 2 and find the maximum sum. Here,  $\max(100+200, 200+300, 300+400) = 700$

**Example 2:** Let the input array be[1, 4, 2, 10, 23, 3, 1, 0, 20], and  $k = 4$

**Output:** 39

**Explanation:** Consider the subarrays consisting of a sliding window of size 4 and find the window having the maximum sum. Here,  $\max(1+4+2+10, 4+2+10+23, 2+10+23+3, 10+23+3+1, 23+3+1+0, 3+1+0+20)=39$

### Solution format

Your solution should be in the following format:

```
1. class Solution():
2.     def maxsumk(self, nums, k):
```

### Strategy

We will start with the simplest strategy that works, but it might be slow. Then, we will refine it and make it efficient.

## Brute force method

First, we will initialize a variable, `maxsum`, to a large negative number. This is because the elements of `nums` could be negative.

We will scan the `nums` array with the index `i`, which goes from 0 to `len(nums)-k+1`

Initialize a variable `curr_sum` to zero.

Add `k` elements from `nums` starting with `i` to `curr_sum`.

Replace `maxsum` with the maximum of `maxsum` and `curr_sum`

## Python code

The following code implements this strategy:

```
1. class Solution():
2.     def maxsumk(self, nums, k):
3.         maxsum = -1e6
4.         for i in range(len(nums)-k+1):
5.             curr_sum = 0
6.             for j in range(i,i+k):
7.                 curr_sum += nums[j]
8.                 maxsum = max(maxsum, curr_sum)
9.         return maxsum
10.
11. #Driver code
12. sol = Solution()
13. nums = [80, -50, 90, 100]
14. k=2
15. print(sol.maxsumk(nums,2))
```

### **Output:**

190

Check that  $\max(80-50, -50+90, 90+100) = 190$

## Complexity Analysis

The complexity analysis is as follows.

### Time complexity:

Let the length of the array be  $n$ . The outer loop runs  $n$  times and the inner loop runs  $k$  times. So, the time complexity is  $O(nK)$ . If  $k$  is constant, the time complexity is  $O(n)$ . But, if we make  $k$  as a fraction of  $n$ , say  $f*n$ , then the time complexity is  $O(n*f*n)$ , that is,  $O(n^2)$ .

How about plotting the time complexity? Let us enclose the above solution in the time measurement loop that we discussed in the *Section 1.10*.

### Python code

The following code adds the time measurement to the previous code:

```
1. import timeit
2. import random
3. from matplotlib import pyplot as plt
4. class Solution():
5.     def maxsumk(self, nums, k):
6.         maxsum = -1e6
7.         for i in range(len(nums)-k+1):
8.             curr_sum = 0
9.             for j in range(i, i+k):
10.                 curr_sum += nums[j]
11.         maxsum = max(maxsum, curr_sum)
12.     return maxsum
13.
14. sol = Solution()
15. nums = [80, -50, 90, 100]
16. k=2
17. #print(sol.maxsumk(nums,2))
18. su=''''
19. from __main__ import sol,nums,k;
20. '''
21. x=[]
22. y=[]
```

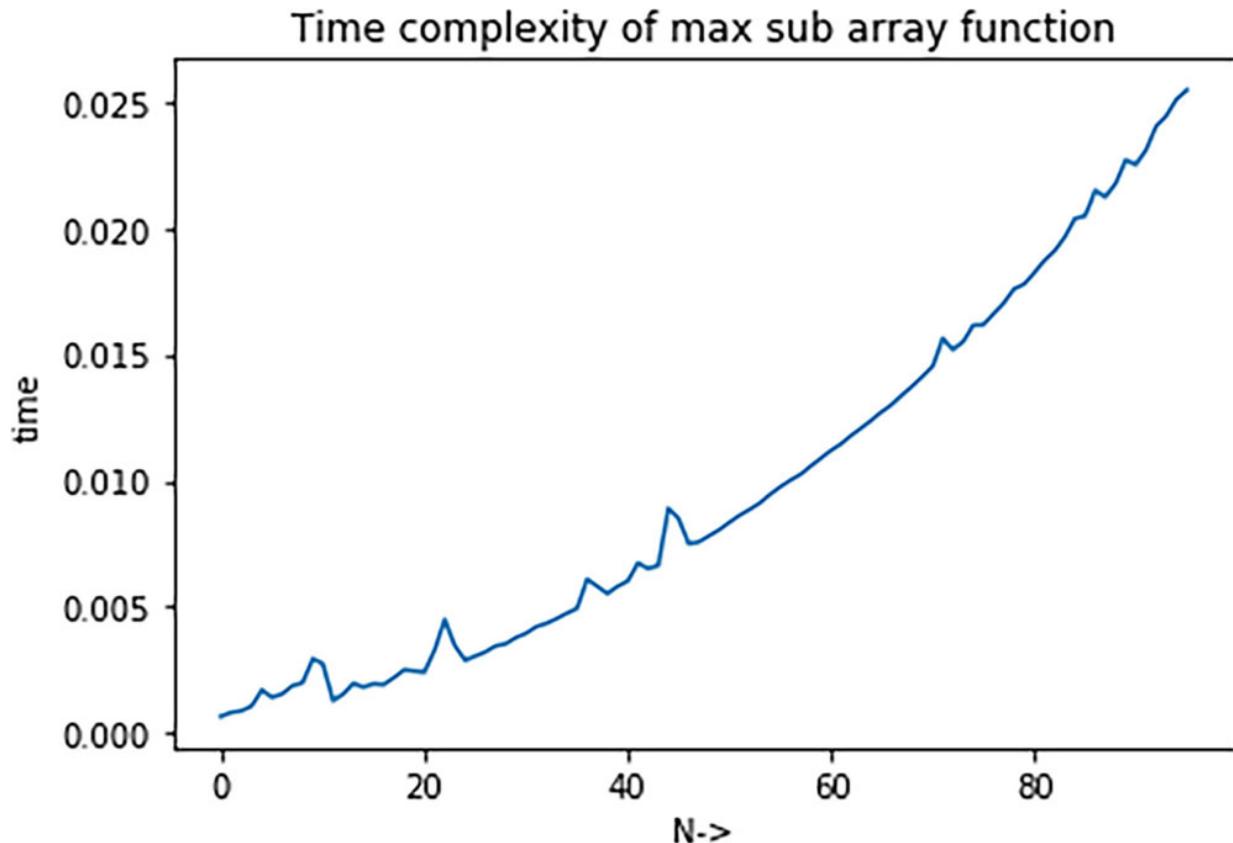
```

23. for i in range(4,100):
24.     nums=[]
25.     k=i//2           #Make k dependent on n
26.     for j in range(i):
27.         nums.append(random.randint(3,10))
28.     #print(nums)
29.

t=timeit.timeit('sol.maxsumk(nums,k)',setup=su,number=100)
30.     x.append(i)  31. y.append(t)
32. plt.plot(x,y)
33. plt.xlabel("N->")
34. plt.ylabel("time")
35. plt.title("Time complexity of max sub array function")

```

The resulting graph is shown in the following figure. It exhibits the  $O(n^2)$  time complexity.



**Figure 1.14:** Time complexity graph for the max subarray function

This is the familiar  $O(n^2)$  graph.

## Efficient method

The outer loop in the above program runs  $n$  times and there is no escape from it. But we can make the inner loop more efficient. Take a look at the second example.

**Example 2:** Input: arr = [1, 4, 2, 10, 23, 3, 1, 0, 20]

```
k = 4
max(1+4+2+10, 4+2+10+23, 2+10+23+3, 10+23+3+1, 23+3+1+0,
3+1+0+20)=39
```

Observe that in each inner loop iteration, we do four additions. We have already done three of them in the previous iteration. We need not start from the scratch in each iteration. We can reuse the result from the previous iteration by subtracting the  $nums[i-1]$  element from the sum and adding the  $nums[i+k-1]$  element. We need to add all the  $k$  members only in the first iteration. The number of additions reduces from  $k$  to 2. It also means that the new method will be beneficial only if  $k$  is greater than 2. Anyway, the time complexity of the inner loop has reduced from  $O(n)$  to  $O(1)$ .

## Python code

The following code implements this strategy:

```
1. class Solution():
2.     def maxsumk(self, nums, k):
3.         maxsum = -1e6
4.         curr_sum = 0
5.         for j in range(k):
6.             curr_sum += nums[j]
7.         for i in range(1, len(nums)-k+1):
8.             curr_sum = curr_sum - nums[i-1] + nums[i+k-1]
9.             maxsum = max(maxsum, curr_sum)
10.        return maxsum
11. sol = Solution()
```

```
12. nums = [80, -50, 90, 100]
13. k=2
14. print(sol.maxsumk(nums, 2))
```

It will be interesting to see the time complexity plot and satisfy ourselves that it is really  $O(n)$ .

## Python code

This code embeds the previous code in a measurement loop:

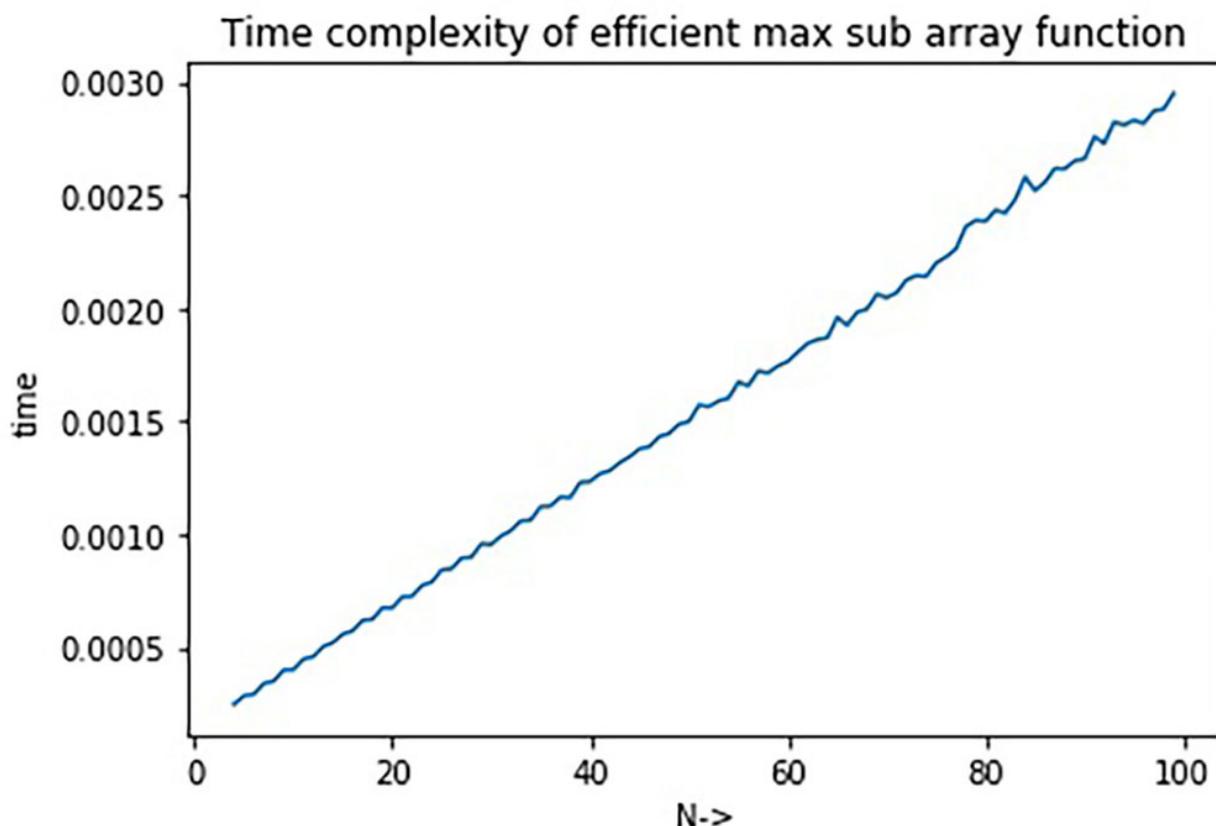
```
1. import timeit
2. import random
3. from matplotlib import pyplot as plt
4. class Solution():
5.     def maxsumk(self, nums, k):
6.         maxsum = -1e6
7.         curr_sum = 0
8.         for j in range(k):
9.             curr_sum += nums[j]
10.            for i in range(1, len(nums)-k+1):
11.                curr_sum = curr_sum - nums[i-1] + nums[i+k-1]
12.                maxsum = max(maxsum, curr_sum)
13.        return maxsum
14. sol = Solution()
15. nums = [80, -50, 90, 100]
16. k=2
17. #print(sol.maxsumk(nums, 2))
18. su=''''
19. from __main__ import sol,nums,k;
20. '''
21. x=[]
22. y=[]
23. for i in range(4,100):
24.     nums=[]
25.     k=i//2
```

```

26.     for j in range(i):
27.         nums.append(random.randint(3,10))
28.         #print(nums)
29.
t=timeit.timeit('sol.maxsumk(nums,k)',setup=su,number=100)
30.     x.append(i)
31.     y.append(t)
32. plt.plot(x,y)
33. plt.xlabel("N->")
34. plt.ylabel("time")
35. plt.title("Time complexity of efficient max sub array
function")

```

The resulting graph is shown in the following figure. It exhibits the  $O(n)$  time complexity. Ignore the kinks in the graph:



**Figure 1.15:** Time complexity graph for the efficient max subarray function

## 1.14 Question 10 – What is the integer part of the square root of a given integer?

Here we are writing a function that returns only the integer part of the square root of a number without using any library function. It presents an unusual application of the binary search technique. This problem tests your general thinking capacity.

### Problem statement

Find the square root of a given integer. If the result is a fraction, truncate it to the floor. Do not use any library function.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def mySqrt(self, x: int) -> int:
```

### Strategy

Since we are not allowed to use the library function *sqrt*, we will do it in a reverse manner, that is, find the squares of various numbers and see if the square matches the given number. We will first use a linear search, and then a binary search.

### Approach 1 – Linear search

This is basically the brute force approach. We will try out all the numbers starting from 0 till the square of the trial number equals or exceeds the given number.

### Python code

The following code implements the linear search strategy. The driver code tests the function for integers in the range 1 to 9:

```
1. class Solution:
```

```

2.     def mySqrt(self, x: int) -> int:
3.         # Base cases
4.         if (x == 0 or x == 1):
5.             return x
6.         # Starting from 1, try all numbers until
7.         # i*i remains less than to x.
8.         i = 1
9.         while (i*i < x):i += 1
10.            return i if i*i == x else i-1
11. #Driver code
12. sol=Solution()
13. for i in range(1,10):
14.     print(i,sol.mySqrt(i))

```

## **Output:**

```

1 1
2 1
3 1
4 2
5 2
6 2
7 2
8 2
9 3

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

As the loop runs  $n$  times, the time complexity is  $O(n)$ .

### **Space complexity:**

As we do not require any additional space, the space complexity is  $O(1)$ .

## **Approach 2 – Binary search**

We will set a search window of the indices start and end. In each iteration, we will find the midpoint of the search window. Now, there are three possibilities:

1.  $mid * mid = x$ . We have found the answer.
2.  $mid * mid < x$ . If  $(mid+1)(mid+1) > x$ . mid is the answer,  
Otherwise, remove the lower half of the window.
3.  $mid * mid > x$ . Remove the upper half of the window.

## Python code

The following code implements this strategy. The driver code from the previous program can be used:

```
1. class Solution:  
2.     def mySqrt(self, x) :  
3.  
4.         # Base cases  
5.         if (x == 0 or x == 1) :  
6.             return x  
7.         # Do Binary Search for integer square root  
8.         start = 1  
9.         end = x  
10.        while (start <= end) :  
11.            mid = (start + end) // 2  
12.  
13.            # If x is a perfect square  
14.            if (mid*mid == x) :  
15.                return mid  
16.  
17.            # when mid^2 is smaller than x, check if (mid+1)^2  
>x  
18.            if (mid * mid < x) :  
19.                if (mid+1)*(mid+1) > x: return mid  
20.                start = mid + 1  
21.
```

```
22.         else :
23.
24.             # If mid*mid is greater than x
25.             end = mid-1
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Due to the binary search, the time complexity is  $O(\log n)$ .

### **Space complexity:**

As we don't need any extra space, the space complexity is  $O(1)$ .

## 1.15 Question 11 – What are the first and last positions of a target number in a given sorted array?

This is a variation of the simple search problem. Consider that the array elements may not be unique, that is, it may contain multiple copies of a number. After sorting, these copies will be placed together. A simple search technique will hit any one of them randomly. But we want to find the beginning and the end of the identical sequence.

### Problem statement

Given an array of integers, `nums`, sorted in an ascending order, find the first and last position of a given target value. Note that the target value may occur in the array only once, or many times, or not at all. If the target value is not found in the array, then you should return `[-1, -1]`. Let us illustrate this with two examples:

#### **Example 1:**

**Input:** Let the array be `[5,6,6,9,9,11]`, and target = 9.

**Output:** `[3,4]`

**Explanation:** 9 is present in the array from positions 3 to 4.

#### **Example 2:**

**Input:** Let the array be [5,6,6,7,7,10], and target = 2.

**Output:** [-1,-1]

**Explanation:** 2 is not present in the array.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def searchRange(self, nums: List[int], target: int) ->  
List[int]:
```

## Strategy

We will present both the linear search and the binary search techniques. The reader is encouraged to try out the time measurement technique presented in *Section 1.10*.

## Linear search (Brute force)

We will start with the simplest strategy. Scan the `nums` array till you find the target. Note the starting index. If you don't find the target by the end, return [-1,-1]. Continue scanning till you find an array element that is not the target. Now, the index `end` should be  $i-1$ . If a non-target is not found till end, `end = len(nums)-1`.

We should return `[start,end]`.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def searchRange(self, nums: List[int], target: int) ->  
List[int]:  
3.         strt = end = -1  
4.         for i in range(len(nums)):  
5.             if nums[i] == target:  
6.                 strt = i
```

```

7.             break
8.         if strt == -1:
9.             return [-1, -1]
10.            #We came here because start was found
11.            for i in range(strt, len(nums)):
12.                if nums[i] != target:
13.                    end = i-1
14.                break
15.            end = len(nums)-1
16.        return [strt, end]
17. #Driver code
18. sol = Solution()
19. nums=[5, 7, 7, 8, 9, 9]
20. target =8
21. print(sol.searchRange(nums, target))

```

### **Output:**

[3, 3]

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Let the length of nums be  $n$ . The for loop runs  $n$  times. Hence, the time complexity is  $O(n)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## Binary search

The linear search method was presented only for a start. The moment we see a sorted array, we should think of the binary search. Using the binary search approach, we start with the *left* and *right* pointers at the beginning and the end of the list, and go on pinching the window depending on the value at the index *mid*. If the target appears only once, that would be the end of the story.

But the target may occur multiple times and we need to find the indices of the first and the last occurrence. So, we split the problem into 2 problems:

1. To find the first occurrence, that is,  $\text{nums}[\text{mid}] == \text{target}$ , and  $\text{nums}[\text{mid}-1] != \text{target}$ . We need to handle the case when the  $\text{mid}$  happens to be 0.
2. To find the last occurrence, that is,  $\text{nums}[\text{mid}] == \text{target}$ , and  $\text{nums}[\text{mid}+1] != \text{target}$ . We need to handle the case when  $\text{mid}$  happens to be the  $\text{len}(\text{nums})-1$ .

We will define two auxiliary functions, `getLeft` and `getRight` for this. The main function, `searchRange` simply calls these two functions one after the other:

```
def searchRange(self, nums: List[int], target: int) ->
List[int]:
    return [self.getLeft(nums, target), self.getRight(nums, target)]
```

Let us talk about the `getLeft` function first. Let  $\text{nums} = [5, 7, 7, 9, 9, 10]$ , and  $\text{target} = 7$ .

The array, `nums`, along with the indices are shown in the following table:

Index	0	1	2	3	4	5
<code>nums[index]</code>	5	7	7	9	9	10

**Table 1.2:** Input array

We will set up the *left* and *right* pointers at the beginning and the end of the list. So, in the beginning, *left* is 0. and the *right* is 5. Then, as long as the *left* is less than the *right*, we run this loop.

Find  $\text{mid} = \text{left} + (\text{right} - \text{left}) // 2$ .

Recall that `//` represents the integer division. But why this funny expression? Why not simply use:

$$\text{mid} = (\text{left} + \text{right}) / 2$$

Well, if we are dealing with really big arrays, then `left + right` might result in an integer overflow. The above expression avoids it.

In the example, we get  $mid = (0+5)//2 = 2$ , and  $nums[2] = 7$ .

Now,  $nums[mid]$  may be equal to the target, less than the target, or greater than the target. Let us consider each case.

1.  $Nums[mid] = target$ : We have landed somewhere in the middle of the target trail. If we are standing on the left edge of the  $nums$  array, that is, if  $mid = 0$ , then we have found the start. Return  $mid$ . But if we are not on the edge, that is, we have something on the left side and that is not the target, even then we found the start. Return  $mid$ .

Otherwise, we are somewhere in the middle of the trail and we need to throw out everything to the right of  $mid$ , including  $mid$ , that is, we replace  $right$  with  $mid-1$ .

This is the case in the first iteration of our example as follows:

$$right = mid-1 = 7$$

2.  $nums[mid] > target$ : This means we are positioned to the right of the target trail. So we should throw out everything to the right of  $mid$ , including  $mid$ , that is, we replace  $right$  with  $mid-1$ .
3.  $nums[mid] < target$ : This means we are positioned to the left of the target trail. So we should throw out everything to the left of , including  $mid$ , that is, we replace  $left$  with  $mid+1$ .

Go to the beginning of the loop.

If we complete the loop, that is, the  $right$  and the  $left$  coincide, and still there is no sign of the target, we should return -1.

When we run the above example, the status of variables in each iteration is shown in the following table. After two iterations, the function returns 1 because  $nums[mid-1] = 5 \neq target$ :

Iteration	Left	Right	Mid	Nums[mid]
Initial	0	5	2	7
1	0	1	0	5
2	1	1	1	7

**Table 1.3:** Values of variables in the example

## Python code

The following code implements this strategy:

```
1. def getLeft(self, nums, target):
2.     left = 0
3.     right = len(nums)-1
4.
5.     while(left <= right):
6.         mid = left+(right-left)//2
7.         if(nums[mid] == target):
8.             if(mid-1 >= 0 and nums[mid-1] != target or mid
== 0):
9.                 return mid
10.            right = mid-1
11.        elif(nums[mid] > target):
12.            right = mid-1
13.        else:
14.            left = mid+1
15.
16.    return -1
```

The code for *getRight* is similar and self-explanatory.

## Python code

The complete code is as follows:

```
1. class Solution:
2.     def searchRange(self, nums: List[int], target: int) ->
List[int]:
3.
4.     return[self.getLeft(nums,target),self.getRight(nums,target)]
5.
6.     def getLeft(self, nums, target):
7.         left = 0
8.         right = len(nums)-1
```

```

8.
9.     while(left <= right):
10.         mid = left+(right-left)//2
11.         if(nums[mid] == target):
12.             if(mid-1 >= 0 and nums[mid-1] != target or mid
13. == 0):
14.                 return mid
15.             right = mid-1
16.             elif(nums[mid] > target):
17.                 right = mid-1
18.             else:
19.                 left = mid+1
20.             return -1
21.
22.     def getRight(self, nums, target):
23.         left = 0
24.         right = len(nums)-1
25.
26.         while(left <= right):
27.             mid = left+(right-left)//2
28.             if(nums[mid] == target):
29.                 if(mid+1 < len(nums) and nums[mid+1] !=
target or\
30.                     mid == len(nums)-1):
31.                     return mid
32.                     left = mid+1
33.                     elif(nums[mid] > target):
34.                         right = mid-1
35.                     else:
36.                         left = mid+1
37.
38.             return -1
39. #Driver code

```

```
40. sol = Solution()
41. nums=[5,7,7,9,9,8]
42. target =7
43. print(sol.searchRange(nums,target))
```

### **Output:**

[1, 2]

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Because of the binary search, the time complexity is  $O(\log n)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## 1.16 Question 12 – What is the position of a search target in a 2D matrix?

This is an extension of the search problem in the linear array to a search problem in a 2D matrix. If we arrange the elements of the matrix in a linear array by row-wise scanning, it is a sorted array. We can either apply 1-D techniques to this array or device special 2-D techniques.

### Problem statement

Write a program that will search a given the target in an  $m \times n$  matrix of integers. Each row of the matrix is sorted in ascending order and the first column of every row is greater than the last column of the previous row. If the target is found, return true, else return false.

### **Example:**

```
m= [
    [1,    2,    5,    7],
    [10,   11,   16,   20],
    [23,   30,   34,   50]
```

```
]  
Target = 5: Output = True  
Target = 31: Output = False
```

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def searchMatrix(self, matrix: List[List[int]], target:  
int) -> bool:
```

## Strategy

We will present the following three approaches:

1. Linear search considering the matrix as a 1D array
2. 2D linear search
3. 2D binary search

## Approach 1 – Linear search

This is the most basic but easiest strategy. Simply run a 2 level nested loop covering every element of the matrix. The Python code is not given here because this approach is not worth considering.

## Complexity Analysis

### **Time complexity:**

Because of the two nested loops, the time complexity is  $O(mn)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## Approach 2 – 2D Linear search

This is better than the linear search because it takes advantage of the sorted nature of the matrix. We will start from the top right corner, that is,  $row = 0$ , and  $col = width-1$ . We will go on shrinking the window till we hit the target.

In each iteration, we will compare the target with the  $\text{matrix}[\text{row}][\text{col}]$ . There are 3 possibilities of the comparison. The action taken for each is as follows:

1. If  $\text{matrix}[\text{row}][\text{col}]$  is equal to the target, we have found the answer.
2. If  $\text{matrix}[\text{row}][\text{col}]$  is greater than the target, reduce the col by 1 and continue.
3. If  $\text{matrix}[\text{row}][\text{col}]$  is less than the target, increase the row by 1 and continue.

If we don't hit the target till the end, return false.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
4.         if not matrix or len(matrix) < 1 or len(matrix[0]) <
1:
5.             return False
6.         row, col = 0, len(matrix[0]) - 1
7.         while col >= 0 and row <= len(matrix) - 1:
8.             if matrix[row][col] == target:
9.                 return True
10.            elif matrix[row][col] > target:
11.                col -= 1
12.            else:
13.                row += 1
14.        return False
15. #Driver code
16. sol=Solution()
17. m= [
18.     [1,    3,    5,    7],
```

```
19.    [10, 11, 16, 20],  
20.    [23, 30, 34, 50]  
21.]  
22. print(sol.searchMatrix(m,3))  
23. m = [  
24.    [1, 3, 5, 7],  
25.    [10, 11, 16, 20],  
26.    [23, 30, 34, 50]  
27.]  
28. print(sol.searchMatrix(m,13))
```

### **Output:**

True  
False

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Because of a single loop, the time complexity is  $O(m + n)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## Approach 3 – 2D Binary search

This is the best approach. We will first find the row by executing binary search on the first column, and then find the target by executing the binary search on the selected row.

## Python code

The following code implements this strategy. The driver code from the previous program can be used:

```
1. from typing import List  
2. class Solution:
```

```

3.     def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
4.         if len(matrix) == 0: return False
5.         row = matrix[0]
6.         if len(row) == 0: return False
7.         #Create an array from first column of every row
8.         col0 = []
9.         for i in range(len(matrix)): col0.append(matrix[i][0])
10.        #print(col0)
11.        #Do binary search to find a row <= target
12.        low = 0; high = len(col0) - 1
13.        while low <= high:
14.            mid = (low+high)//2
15.            if col0[mid] == target: break
16.            if col0[mid] < target:
17.                if mid == len(col0)-1: break
18.                if col0[mid+1] > target: break
19.                else: low = mid+1
20.            else:
21.                high = mid-1
22.            row = matrix[mid]
23.            print(row)
24.            #Now search target in row
25.            low = 0; high = len(row) - 1
26.            while low <= high:
27.                mid = (low+high)//2
28.                if row[mid] == target: return True
29.                if row[mid] < target: low = mid+1
30.                else: high = mid-1
31.        return False

```

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:**

Because of the binary search, the time complexity is  $O(\log m + \log n)$ .

### Space complexity:

We don't need any extra space. So, the space complexity is  $O(1)$ .

## 1.17 Question 13 – How will you convert an integer into a roman numeral?

Converting an integer into a roman numeral string is an excellent example of text processing.

### Problem statement

Convert an integer into its roman notation.

Roman numerals are represented by seven alphabets: I, V, X, L, C, D, and M. Their values are tabulated below in [Table 1.4](#):

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

*Table 1.4: Roman symbol values*

The roman system is not positional. The total value of the number is the sum of the symbols. A symbol can be repeated up to 3 times. The symbols are normally written in decreasing order.

For example, MDC = 1000 + 500 + 100 = 1600

But, if a smaller symbol is written before a larger one, the value of the smaller symbol is negative.

For example, IV = -1 + 5 = 4

Example 1: 3 -> “III”

Example 2: 4 -> “IV”

Example 3: 9 -> “IX”

Example 4: 58 -> “LVIII”

Example 5: 1994 -> “MCMXCIV”

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def intToRoman(self, num: int) -> str:
```

## Strategy

A smaller symbol appearing before a bigger one becomes negative. There are only six possibilities for this. To take care of them, we will consider a two-letter symbol as a token and add 6 more entries to the following table:

Symbol	Value
IV	4
IX	9
XL	40
XC	90
CD	400
CM	900

*Table 1.5: Two-letter symbol values*

The table will be created as an array of tuples, where each tuple will be a (value, symbol) pair. We will combine the single letter and double letter symbols into a single array, *v*. The tuples in the combined array are arranged in decreasing order of their value.

We will traverse the array, *v* with the pointer *i*. Any symbol can be repeated up to 3 times. The number of repetitions of the symbol is `num // v[i][0]`. For example, number 31 is converted to XXXI. So, X is repeated thrice. At

each repetition, the symbol is added to the output string *roman\_str*, and the value is subtracted from *num*. When *num* becomes 0, *roman\_str* contains the answer.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def intToRoman(self, num: int) -> str:
3.         v = [(1000,"M"),(900,"CM"),(500,"D"),(400,"CD"),
4.               (100,"C"),(90,"XC"),(50,"L"),(40,"XL"),
5.               (10,"X"),(9,"IX"),(5,"V"),(4,"IV"),(1,"I")]
6.         roman_str = ''
7.         i = 0
8.         while num > 0:
9.             m = v[i]
10.            for k in range(num // v[i][0]):
11.                roman_str += v[i][1]
12.                num -= v[i][0]
13.                i += 1
14.        return roman_str
15.
16.
17. print(Solution().intToRoman(1))
18. print(Solution().intToRoman(2020))
```

### **Output:**

```
I
MMXX
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The loop runs only once and its execution time is independent of the length of the string. Hence, the time complexity is  $O(n)$ .

**Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

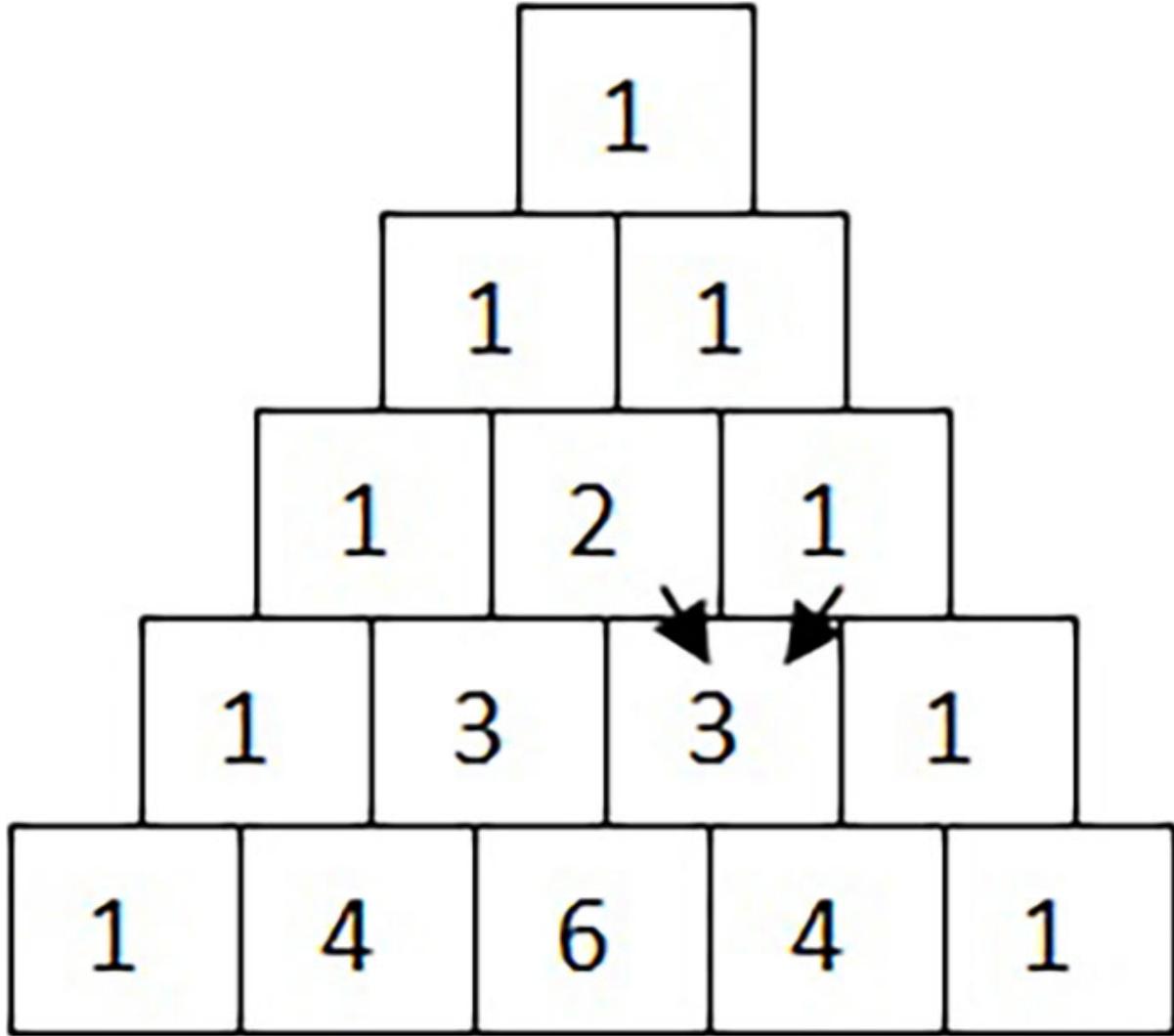
### 1.18 Question 14 – How will you construct Pascal's triangle?

This mathematics oriented problem tests your general programming skills. Pascal's triangle shows the coefficients in a binomial expansion. Consider the following cases:

$$(x + 1)^2 = 1 + 2x + x^2$$

$$(x + 1)^3 = 1 + 3x + 3x^2 + x^3$$

$$(x + 1)^4 = 1 + 4x + 6x^2 + 4x^3 + x^4$$



**Figure 1.16:** Pascal's triangle of order 4

We can arrange the coefficients in a triangle and add 1 and 11 at the top as shown in [Figure 1.16](#).

An  $n^{\text{th}}$  order Pascal's triangle has  $n$  rows. The first and the last number of each row is 1. That means row 0 = [1] and row 1 = [1,1]. The middle elements ( $j$ ) of  $n^{\text{th}}$  row are constructed according to the following formula:

$$\text{row}[n][j] = \text{row}[n-1][j-1] + \text{row}[n-1][j]$$

That is, the element  $j$  in the  $n^{\text{th}}$  row is the sum of the elements to the left and right of  $j$  in the row above. In [Figure 1.16](#), observe the arrows that show that  $3 = 2 + 1$ .

It is possible to compute the  $j^{\text{th}}$  element of the  $n^{\text{th}}$  row directly as  ${}^n C_j$ . However, Pascal's triangle is a numerically efficient way of enumerating all the coefficients.

## Problem statement

Write a program to construct Pascal's triangle of order  $n$ .

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def generate(self, num_rows):
```

## Strategy

First, we will create an empty list, named `answer`. Then, we will run a loop with an iterator,  $n$  whose values will go from 0 to  $\text{num\_rows} - 1$ . In each iteration, we will first create a list, named `row` of length  $n+1$  consisting of all the zeroes. Then, we will make its first and last elements 1. The previous row is stored in `answer[n-1]`. We will fill the middle values of row  $n$  according to the formula:

```
row[j] = answer[n-1][j-1] + answer[n-1][j]
```

When all the elements are filled, we will append the row to the answer.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def generate(self, num_rows):  
3.         answer = [] # Collection of rows  
4.  
5.         for n in range(num_rows):  
6.             #Create initial version of nth row  
7.             # The first and last row elements are always 1.
```

```

8.          # Let middle elements be 0
9.          row = [0 for dummy in range(n+1)]
10.         row[0], row[-1] = 1, 1
11.         #Now fill middle elements
12.         # Each element of nth row is equal to the sum
of the elements
13.         # above-and-to-the-left and above-and-to-the-
right.
14.         #Let j span the nth row
15.         for j in range(1, len(row)-1):
16.             row[j] = answer[n-1][j-1] + answer[n-1][j]
17.             #nth row is ready. append it to answer
18.             answer.append(row)
19.
20.         return answer
21. sol = Solution()
22. print(sol.generate(6))

```

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** As there are two nested loops, the time complexity is  $O(n^2)$ .

**Space complexity:** As we are creating  $n$  arrays, the space complexity is  $O(n^2)$ .

## Conclusion

We began with an explanation of the two important aspects of an algorithm, namely the time complexity, and the space complexity. We also studied the techniques for practically measuring the time complexity. We studied 14 problems which sharpened our skills in manipulating strings and lists. We studied the binary search technique applied to various situations. We studied how to think of a strategy to solve a problem and convert it into code. We also studied how we can solve a problem using various approaches and trade the space complexity with the time complexity.

In this chapter, we concentrated only on the array data structure. In the next chapter, we will study questions based on two more data structures, namely linked lists and stacks.

## Points to Remember

- The program efficiency is expressed in Big O notation
- Aggregate data objects:
  - **Tuple:** `a = (1,2,3)`
  - **List:** `a = [1,2,3]`
  - **String:** `a = “abcd”`
- The time complexity of a few common algorithms is:
  - Two nested loops:  $O(n^2)$
  - Sorting:  $O(n \log n)$
  - Binary search:  $O(\log n)$
  - Linear search:  $O(n)$
  - Fibonacci numbers:  $O(2^n)$
- Use `NumPy.matmul` to multiply the matrices. If you use `*` operator, it results in the element by element multiplication.
- Typing hints like `def func(i:int) -> bool` are used to improve readability.

## MCQs

1. Which of the following commands will create a list?
  - A. `Mylist = list([1, 2, 3])`
  - B. `Mylist = []`
  - C. `Mylist = list()`
  - D. All of the above
2. What is the output when we execute the list (“python”) at the Python shell?

- A. ['p', 'y', 't', 'h', 'o', 'n']  
B. python  
C. [python]  
D. ['python']
3. What is the output if we execute the following commands at the Python prompt?
- ```
a=[1,2,3,4,5]
a[-1]
```
- A. Error  
B. 1  
C. 5  
D. None
4. What will be the output of the following command?
- ```
print('pq'.zfill(5))
```
- A. pq  
B. 000pq  
C. pq000  
D. 0pq00
5. What is the time complexity of a binary search algorithm?
- A. O(n)  
B. O(log n)  
C. O(n log n)  
D. O( $2^n$ )
6. What is the output of the following commands?

```
Mylist = [None] * 10
print(len(Mylist))
```

- A. 10  
B. 0

C. None

D. Error

7. What is the output of the following commands?

```
Mylist=[“abcd”, [1, 2, 3, 4]]  
print(Mylist[0][1], Mylist[1][3])
```

A. a 1

B. b 4

C. a 3

D. Error

8. What is the output of the following commands?

```
Mylist=['c', 'java', 'python']  
max(Mylist)
```

A. c

B. python

C. java

D. Error

9. What is the output of the following commands?

```
Mylist =[1, 2, 3, 4, 5]  
Mylist[-4:-1]
```

A. [2,3,4]

B. [2,3,4,5]

C. [4:3:2]

D. Error

10. What is the output of the following commands?

```
Mylist = [x+y for x in ['a', 'b'] for y in ['c', 'd']]  
Mylist
```

A. ['abcd']

- B. [‘ab’, ‘cd’]
- C. [‘ac’, ‘ad’, ‘bc’, ‘bd’]
- D. Error

## Answers to MCQs

**Q1:** D

**Q2:** A

**Q3:** C

**Q4:** B

**Q5:** B

**Q6:** A

**Q7:** B

**Q8:** B

**Q9:** A

**Q10:** C

## Questions

1. What is time complexity and space complexity?
2. Why is the binary search better than the linear search?
3. What is the sliding window approach?
4. What is recursive calling?
5. Why is the time measurement not accurate?

## Key Terms

Time complexity, space complexity, linear search, binary search, sliding window, brute force method.

## CHAPTER 2

### Linked Lists and Stacks

The linked list is an important data structure that has wide applications in computer science. Some of the major applications are used to store the addresses of free memory blocks for dynamic memory allocation, represent sparse matrices, represent graphs in adjacency list form, and maintain a directory of names that has frequent insertions and deletions.

Insertions and deletions in a linked list can be done with  $O(1)$  time complexity. But the search operation needs  $O(n)$  time complexity since we cannot perform a binary search.

#### Structure

We assume that the reader is already familiar with the linked lists. However, we will review the basic concepts and perform some exercises on the linked list operations like creation, display, addition, and deletion of nodes. Then, we will move on to various questions based on the applications of linked lists. As the test questions are based on singly linked lists, we have covered singly linked lists here. We will be covering the topics in the following order:

- Basics of linked lists
- Question 15: How will you merge two sorted lists?
- Question 16: How will you detect a cycle in a linked list?
- Question 17: How will you reverse a linked list?
- Question 18: How will you add two linked lists?
- Question 19: How will you remove the  $n^{\text{th}}$  node from the right?
- Question 20: How will you create odd and even linked lists?
- Question 21: How will you evaluate reverse Polish postfix expression?
- Question 22: How will you achieve the minimum implementation of a stack using a list?

## Objectives

After studying this unit, you will be comfortable with the representation of linked lists in memory, list operations, and applications.

### 2.1 Basics of Linked Lists

Before moving on to the problems, we will revise some basics.

#### 2.1.1 Node, the basic building block

A linked list is made up of a chain of nodes. For representing a node of a linked list, we need a data structure consisting of two items:

1. **Information:** It is also called value or data. In most of the examples which we will discuss, the information consists of a single item, usually an integer. But in general, it can be any other data type or even a collection of the data.
2. **Link:** This is a pointer to the next item on the list.

The first node in the list is called the head, the last node is called the tail, and its link is null. In C or Java, null is represented by 0. In Python, we use a special object None.

For solving questions on the linked lists, we will use the `ListNode` class defined below. The names of variables are only indicative. They may change from problem to problem:

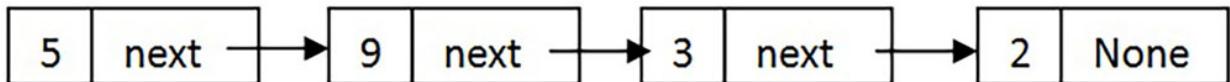
```
1. class ListNode:  
2.     def __init__(self, x): #Constructor  
3.     self.val = x          #Initialize value with parameter x  
4.     self.next = None      #Next pointer will be None by  
default
```

Usually, the class `node` is assumed to be predefined in the test environment and you need to write only the solution class. But for testing and debugging the code before submitting it, you need to include it in your code. For testing, you also need to create lists in the driver code to act as test inputs.

Python does not have any built-in support for the linked lists. We will make our own helper functions for this as described below.

## 2.1.2 Creating and displaying a linked list

Small lists can be created manually as shown below. Refer to the list shown in the following figure:



*Figure 2.1: Linked list*

The following code creates the above linked list:

```
1. n1=ListNode(5)
2. n2=ListNode(9)
3. n3=ListNode(3)
4. n4=ListNode(2)
5.
6. n1.next = n2
7. n2.next = n3
8. n3.next = n4
```

You can enter the above code in the IDE and build it in memory. But it is very difficult to observe it in a debugger. It will be nice to have a helper function `display` to display the list. It displays the data field from the root to the end. Note that the root need not be the head of the list. It is just the starting point for the list.

## Python code

The following code implements the list display function. The driver code for creation of test list[1,3,5] and the node class definition are also included:

```
1. def display(root):
2.     while (root != None) :
3.         print(root.data, end = " ")
4.         root = root.next
```

Try the following complete program:

```
1. class ListNode:  
2.     def __init__(self, val=0, next=None):  
3.         self.val = val  
4.         self.next = next  
5.  
6.     def display(self):  
7.         while (root != None) :  
8.             print(root.val, end = " ")  
9.             root = root.next  
10.  
11. n1=ListNode(1)  
12. n2=ListNode(3)  
13. n3=ListNode(5)  
14. n1.next=n2  
15. n2.next=n3  
16. n3.next=n4  
17. display(n1)
```

**Output:** 1 3 5

Sometimes we keep a separate pointer called the head which points to the beginning node of the list. Sometimes there is no separate pointer. We simply specify the beginning node directly, like the node *n1* in the above code. Sometimes we also maintain a pointer to the tail. It makes a fast insertion or deletion at the end.

### 2.1.3 Adding a node in the beginning

Suppose you want to add a node named *n4* having value 10 at the beginning of the list created in the previous exercise. First, create the node *n4* and fill its next pointer with *n1*. The display function will be called with *n4* as an argument because the new head will be *n4*.

### Python code

The following lines add node *n4* at the beginning.

```
1. n4=ListNode(10)
2. n4.next=n1
3. display(n4)
```

**Output:** 10 1 3 5

#### 2.1.4 Adding a node at the end

Suppose you want to add a node named *n5* having a value of 20 at the end of the list created in the above exercise. First, create the node *n5*. But, the addition is not straightforward because you do not know the location of the last element. You must first traverse the list starting from the root till you find the next pointer to be `None`. Change that pointer so that it points to the new node.

Thus, we can conclude that adding a node in the beginning has a time complexity of  $O(1)$ , and adding a node in the end has a time complexity of  $O(n)$ .

#### Python code

The following code adds a node *n5* having value 20 at the end of the given list:

```
1. n5=ListNode(20)
2. n=n1
3. while n.next:
4.     n=n.next
5. n.next=n5
6. display(n1)
```

**Output:**

10 1 3 5 20

#### 2.1.5 Creating a list from an array

The above method of creating a list is very cumbersome. It is convenient to create a list from an array. Also, most of the time, the list is given as an array in the test question. We will write a function *MakeList* for this.

## Python code

The following code implements the *MakeList* function to create a list from an array:

```
1. def MakeList(a):
2.     root = ListNode(a[0])
3.     cur = root
4.     for i in range(1, len(a)):
5.         temp = ListNode(a[i])
6.         cur.next = temp
7.         cur = cur.next
8.     return root
9.
10. myList = MakeList([1, 2, 3, 4, 5])
11. display(myList)
```

**Output:** 1 2 3 4 5

## 2.2 Question 15 – How will you merge two sorted lists?

Based on the fundamentals given above, you should be able to answer this question.

### Problem statement

You are given two singly linked lists which are sorted in ascending order. Write a program to merge these lists and produce an output list. Do not produce a separate output list. Rearrange the pointers to make a merged list in-place. Let us illustrate this with an example.

**Example:**

**Input:**

1->2->4,

1->3->4

**Output:** 1->1->2->3->4->4

**Explanation:** We have taken the elements from both the lists and produced the output list in a proper order.

## Solution format

Your solution should be in this format:

```
1. # Definition for singly-linked list.
2. # class ListNode:
3. #     def __init__(self, val=0, next=None):
4. #         self.val = val
5. #         self.next = next
6. class Solution:
7.     def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
ListNode:
```

## Strategy

We will create an object of the *node* class, named *cur*. This will hold the current pointer to the output list. The original value of *cur* will be saved in a variable, named *ans*. The input arguments, *l1* and *l2* initially point to the roots of the input lists. Now we will start a while loop. We will compare the values in *l1* and *l2*, and copy the node having the smaller value in *cur.next*. Then, we will advance the pointers *l1* (or *l2*) to point to the next elements in their respective lists. At the end of the loop, we will replace *cur* with *cur.next*. So, *cur* becomes either *l1* or *l2*, whichever has a smaller value. This will continue as long as both the lists have elements. Eventually, one of them will dry up, or both of them may dry up together. Two more while loops mop up the remaining elements and add them to the output list. Finally, we will return *ans.next*.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. def display(root): #Helper function to print answer
3.     while (root != None) :
```

```
4.         print(root.val, end = " ")
5.         root = root.next
6.
7. class ListNode: #Test environment function
8.     def __init__(self, val=0, next=None):
9.         self.val = val
10.        self.next = next
11. class Solution:
12.     def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
ListNode:
13.         #l1 and l2 are initially roots of the two input
lists
14.         #As we will append the current element to output, we
will advance them
15.         cur = ListNode(0)    #Create an empty node for output
list
16.         ans = cur           #ans starting position is empty
node
17.         while(l1 and l2):   #Execute this till both lists
have elements
18.             if(l1.val>l2.val):
19.                 cur.next = l2 #If l2 is smaller, insert it
in current
20.                 l2 = l2.next  #Advance the pointer for l2
21.             else:
22.                 cur.next = l1 #If l1 is smaller, insert it
in current
23.                 l1 = l1.next  #Advance the pointer for l2
24.                 cur = cur.next #Advance the pointer for
current
25.         while(l1):          #Execute this if l1 is not
finished
26.             cur.next = l1 #Append l1 element to cur
27.             l1 = l1.next  #Advance the pointer for l1
```

```

28.             cur = cur.next
29.
30.         while(l2):           #Execute this if l2 is not
finished
31.             cur.next = l2
32.             l2 = l2.next #Advance the pointer for l2
33.             cur = cur.next#Advance the pointer for
current
34.         return ans.next      #next field of ans is head of
cur list
35. #driver code
36. #Construct first list as [1,3,5]
37. n11 = ListNode(1);n12=ListNode(3);n13=ListNode(5)
38. n11.next = n12;n12.next = n13
39. #Construct second list as [2,4,6]
40. n21 = ListNode(2);n22=ListNode(4);n23=ListNode(6)
41. n21.next = n22;n22.next = n23
42.
43. sol=Solution()
44. merg = sol.mergeTwoLists(n11,n21)
45. display(merg)

```

**Output:** 1 2 3 4 5 6

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If  $n$  is the sum total of the elements in the two lists, that is,  $n = n1 + n2$ , the loops are executed  $n$  times. Hence, the time complexity is  $O(n)$ .

### **Space complexity:**

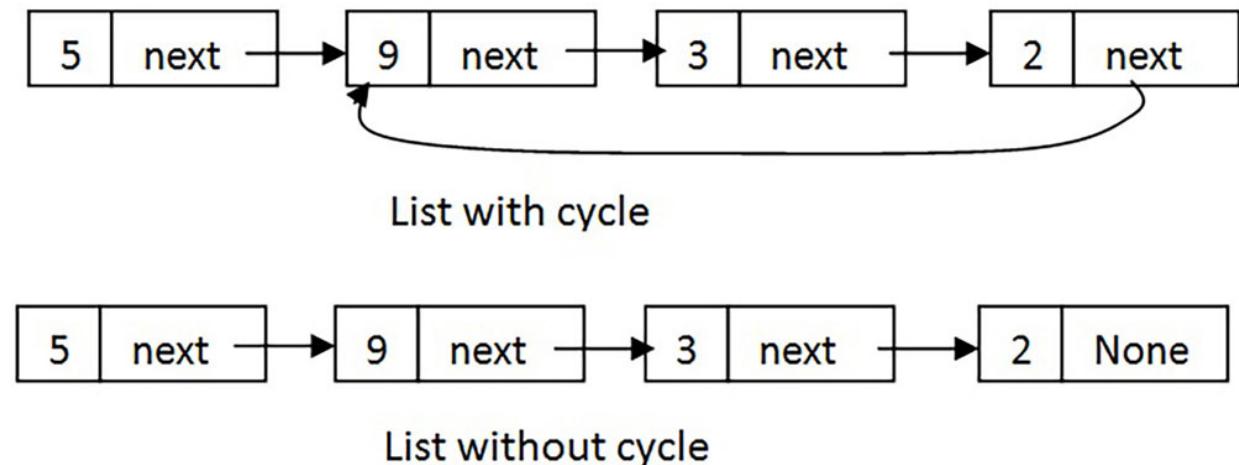
As we are not using any extra space, the space complexity is  $O(1)$ .

## 2.3 Question 16 – How will you detect a cycle in a linked list?

In many applications, we need to detect if there is a cycle in the linked list. For example, there is a problem in which you have to traverse the list from the beginning to the end. If there is a cycle, the program will get locked in an infinite loop. This question will improve your list handling skills.

### Problem statement

Write a program to detect a cycle in the linked list. The following diagram illustrates a list with, and without a cycle:



*Figure 2.2: List with and without cycle*

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def hasCycle(self, head: ListNode) -> bool:
```

### Strategy

We will use a two-pointer approach, slow and fast. Both the pointers start from the head together. But the fast pointer advances two steps at a time. Suppose, there are  $N$  steps before reaching a feedback point (node 9 in [Figure 2.2](#)). When the slow pointer has reached the feedback point, the fast

pointer has moved  $2N$  steps, so, it is somewhere in the cycle. Thereafter, both the pointers move in the cycle. Suppose, the cycle has  $k$  steps. Their relative speed is 1 step per iteration. Therefore, these two pointers should coincide in at the most  $k$  steps. If this happens, we should return `True`.

If the list does not have a cycle, the fast pointer will reach the end. So, we should return `False`. As the fast pointer is jumping two steps, it will skip the end if the number of nodes is odd. To prevent it, we should check the validity of the `fast` and `fast.next`.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. def display(root):
3.     while (root != None) :
4.         print(root.val, end = " ")
5.         root = root.next
6.
7. class ListNode:
8.     def __init__(self, val=0, next=None):
9.         self.val = val
10.        self.next = next
11. class Solution:
12.     def hasCycle(self, head: ListNode) -> bool:
13.         fast = head
14.         slow = head
15.
16.         while slow and fast and fast.next:
17.             fast = fast.next.next
18.             slow = slow.next
19.             if(slow == fast):
20.                 return True
21.         return False
22. #Driver code
```

```
23. #Create list n1->n2->n3->n4->n2
24. n1 =
ListNode(3);n2=ListNode(2);n3=ListNode(0);n4=ListNode(-4)
25. n1.next = n2;n2.next = n3;n3.next = n4;n4.next = n2
26. sol=Solution()
27. print(sol.hasCycle(n1))
```

**Output:** True

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The slow pointer moves  $n-k$  times before entering the cycle, and at the most,  $k$  times in the cycle before exiting. Thus, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are not using any extra space, the space complexity is  $O(1)$ .

## 2.4 Question 17 – How will you reverse a linked list?

This question will enhance your comprehension of lists and develop your manipulation strategies.

### Problem statement

Write a program to reverse a given linked list. Let us illustrate the required operation with help of an example.

#### **Example:**

**Given list:** 1 → 2 → 3 → 4 → NULL

**Reversed list:** 4 → 3 → 2 → 1 → NULL

**Explanation:** The head of the answer list is the tail of the original list, and the elements appear in reverse order.

### Solution format

Your solution should be in the following format:

```

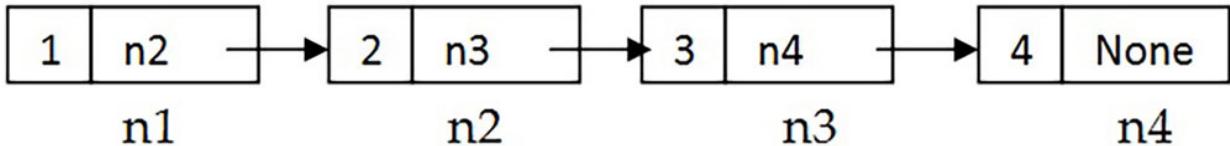
1. class Solution:
2.     def reverseList(self, head: ListNode) -> ListNode:

```

## Strategy

A brute force method would be to read the elements from the given list one by one, and insert them at the beginning of the output list. The time complexity of this method would be  $O(n)$  because insertion, in the beginning, happens with the complexity  $O(1)$ . However, the space complexity would be  $O(n)$ .

We can achieve the space complexity of  $O(1)$  if we simply reverse the direction of the arrows in place. The time complexity remains  $O(n)$ . The initial layout of the list is shown in [Figure 2.3](#):



*Figure 2.3: Initial layout of the list*

The pointer *head* is supplied as the argument. We will use it for traversing the list. The steps for reversing are as follows:

First, the variable *Left* is initialized to *None*.

Then, the following steps are repeatedly executed as long as the *head* is not *None*:

1. Save the current element's next pointer in the variable *Right*.
2. Set the current element's next pointer to the *Left* variable saved in *Step 3* of the previous iteration.
3. Save the current element's pointer in the variable *Left* for use in the next iteration.
4. Move on to the next element on the right by storing the variable *Right* (saved in *Step 1*) in the *head*.

When the *head* reaches the last node, its next pointer (which happens to be *None*) is stored in the *head* and the pointer to the last node is stored in the

variable *Left*. Since, the *head* is null, the program exits from the loop, and the *Left* is the head of the reversed list.

***None* is actually an object of the class *NoneType*. To test if a variable is *None*, we should not use equality operators, == or !=. These may misbehave in some cases. Use the keywords “is” or “is not”.**

## Python code

The following code implements this strategy. The code for displaying which was explained in *Section 2.1.2* is reused.

```
1. def display(root):
2.     while (root != None) :
3.         print(root.val, end = " ")
4.         root = root.next
5.     print('\n')
6. class ListNode:
7.     def __init__(self, val=0, next=None):
8.         self.val = val
9.         self.next = next
10.    class Solution:
11.        def reverseList(self, head: ListNode) -> ListNode:
12.            Left = None           #Initialize left pointer to
null
13.            while(head is not None):#Scan the list with head
(Current)
14.                Right = head.next #Save pointer of element to
right of current
15.                head.next = Left  #Current element's next
pointer = Left
16.                Left = head      #Current becomes Left in next
iteration
17.                head = Right   #Advance current pointer to saved
Right
18.        return Left
```

```
19. #Driver code : Create list [1, 2, 3, 4]
20. n1 =
ListNode(1);n2=ListNode(2);n3=ListNode(3);n4=ListNode(4)
21. n1.next = n2;n2.next = n3;n3.next = n4;n4.next = None
22. display(n1)
23. sol=Solution()
24. r=sol.reverseList(n1)
25. display(r)
```

### **Output:**

```
1 2 3 4
4 3 2 1
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If the list has  $n$  elements, the loop runs  $n$  times. So, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are not using any extra space, the space complexity is  $O(1)$ .

## 2.5 Question 18 – How will you add two linked lists?

Do you know that the maximum value a 32-bit integer can hold is 2,147,483,647, and the maximum value a 64 bit integer can hold is 9,223,372,036,854,775,807? These numbers are great! But, suppose you want to represent the numbers that are arbitrarily large? One way of representing them will be with the linked lists. This question asks you to add two numbers stored in this format.

### Problem statement

You are given two linked lists containing two decimal numbers. The digits are stored in the reverse order, that is, LSD first. Write a program to add the lists and produce an output list in the same format. The lists are non-empty,

but may be of different lengths. Let us illustrate the required process with this example.

### **Example:**

**Input:**  $(1 \rightarrow 3 \rightarrow 5) + (2 \rightarrow 4 \rightarrow 6)$

**Output:**  $3 \rightarrow 7 \rightarrow 1 \rightarrow 1$

**Explanation:** The first number is 531, and the second number is 642.

Check that  $531 + 642 = 1173$ .

### **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def addTwoNumbers(self, l1: ListNode, l2: ListNode) ->  
ListNode:
```

### **Strategy**

We will create an empty node and add it to the answer list. We will create a variable *carry* to store the carry resulting from the addition so that it can be used in the next iteration. We will initialize *carry* to 0. As long as the input lists are not finished, we will carry out the following steps:

1. Add the previous *carry* and the two LSD's to get sum.
2. Sum modulo 10 gives the answer digit. Store it in a new node and add it to the answer list.
3. Advance l1, l2, and pointers.
4. Integer division of sum by 10 gives *carry* to be used in the next iteration.

If the last iteration results in a *carry*, we will create a new node and add it to the answer.

### **Python code**

The following code implements this strategy:

```

1. def display(root):
2.     while (root != None) :
3.         print(root.val, end = " ")
4.         root = root.next
5.     print('\n')
6. class ListNode:
7.     def __init__(self, val=0, next=None):
8.         self.val = val
9.         self.next = next
10. class Solution:
11.     def addTwoNumbers(self, l1: ListNode, l2: ListNode) ->
ListNode:
12.         ans = ListNode(None)
13.         pointer = ans           #Pointer in answer string
14.         carry = 0;             #To begin with, carry is
0
15.         while(l1!=None or l2!=None):#Keep looping l1 and l2
are not finished
16.             sum = carry #Initialize sum with last
iteration's carry
17.             if(l1!=None):
18.                 sum+=l1.val #If l1 is not finished, add
l1.val to sum
19.                 l1 = l1.next #Advance l1 to next element
20.             if(l2!=None):
21.                 sum+=l2.val #if l2 is not finished, add
l2.val to sum
22.                 l2 = l2.next #Advance l2 to next element
23.                 carry = sum//10      #integer division
24.                 pointer.next = ListNode(sum%10)#create new node
with sum modulo 10
25.                 pointer = pointer.next #Advance the answer
pointer

```

```

26.         if carry:                      #If last addition creates
carry,
27.             pointer.next = ListNode(carry)#Create one more
node and add it
28.         return ans.next
29. #driver code
30. n11 = ListNode(1);n12=ListNode(3);n13=ListNode(5)
31. n11.next = n12;n12.next = n13
32. n21 = ListNode(2);n22=ListNode(4);n23=ListNode(6)
33. n21.next = n22;n22.next = n23
34. sol=Solution()
35. add = sol.addTwoNumbers(n11,n21)
36. display(add)

```

**Output:** 3 7 1 1

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If the longest list has  $n$  elements, the loop runs  $n$  times. So, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are not using any extra space, the space complexity is  $O(1)$ .

## 2.6 Question 19 – How will you remove the $n^{\text{th}}$ node from the right?

This question will enhance your list of traversal skills.

### Problem statement

From a given a linked list, remove the  $n^{\text{th}}$  node from the end of the list and return the head of the resulting list. Let us illustrate this with the help of an example.

### **Example:**

Given linked list: 1->2->3->4->5->6->7 and n = 3.

**Output:** 1->2->3->4->6->7, 1

The third node from the end is 5. After removing it, the linked list becomes 1->2->3->4->6->7. Its head is 1.

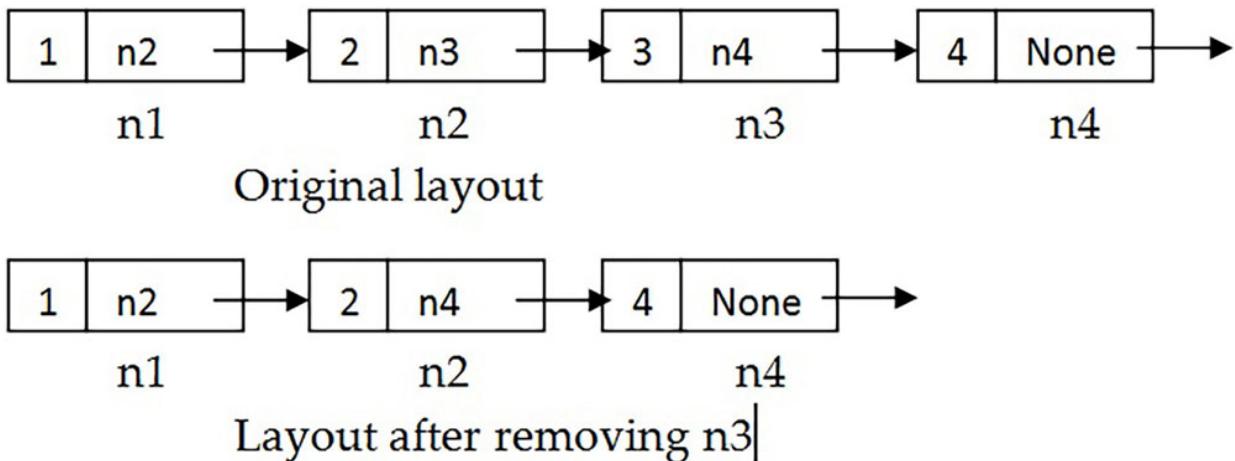
## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def removeNthFromEnd(self, head: ListNode, n: int) ->  
ListNode:
```

## Strategy

Consider the list 1->2->3->4, and n=2. The node second from the right is  $n_3$ . [Figure 2.4](#) shows the original layout of the list and the layout after removing  $n_3$ :



*Figure 2.4: Original and changed layouts of the list*

In order to remove  $n_3$ , the next field of  $n_2$  has to be set to  $n_4$ . But before that, we have to locate  $n_2$ , which happens to be the second node from the right. Since this is a singly linked list, there is no way of tracing it back from the end. But, we can solve this problem using a sliding window approach. We will use two pointers with a difference of  $n$  elements between them. Then, we let them traverse the list at the same rate. When the first pointer's

next field is *None*, the second pointer will be pointing to the element to the left of the element to be removed. Simply change its next pointer.

If  $n = 1$ , we can use this function. Remove the tail with the time complexity  $O(n)$ .

## Python code

The following code implements this strategy:

```
1. def display(root):
2.     while (root != None) :
3.         print(root.val, end = " ")
4.         root = root.next
5.     print('\n')
6. class ListNode:
7.     def __init__(self, val=0, next=None):
8.         self.val = val
9.         self.next = next
10.    class Solution:
11.        def removeNthFromEnd(self, head: ListNode, n: int) ->
ListNode:
12.            ans = ListNode(0) #Dummy node as head of answer
list
13.            first = ans
14.            second = ans
15.            ans.next = head #Answer points to dummy
16.            #Advance only first pointer by n+1 steps
17.            for i in range(n+1):
18.                first = first.next
19.            #Advance both pointers till the first hits end
20.            while(first is not None):
21.                first = first.next
22.                second = second.next
23.            #Skip the second.next element
24.            second.next = second.next.next
```

```

25.         return ans.next
26. #Driver code
27. n1 =
ListNode(1);n2=ListNode(2);n3=ListNode(3);n4=ListNode(4);n5=
ListNode(5)
28. n1.next = n2;n2.next = n3;n3.next = n4;n4.next=n5;n5.next =
None
29. display(n1)
30. sol=Solution()
31. r=sol.removeNthFromEnd(n1,2)
32. display(r)

```

### **Output:**

```

1 2 3 4 5
1 2 3 5

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Suppose, the list has  $m$  elements. The first loop (*for*) runs  $n$  times. The second loop (*while*) runs  $m-n$  times. So, the time complexity is  $O(m)$ ,

### **Space complexity:**

As we are not using any extra space, the space complexity is  $O(1)$ .

## **2.7 Question 20 – How will you create odd and even linked lists?**

This is an exercise for improving your list manipulation skills.

### **Problem statement**

Rearrange a singly linked list such that all the odd nodes are grouped together, followed by the even nodes. Here, we are talking about the node number and not the value in the nodes. The rearrangement should be done in place. Let us illustrate this with the help of an example.

## Example:

**Input:** 22->21->23->25->26->24->27->NULL

**Output:** 22->23->26->27->21->25->24->NULL

**Explanation:** The odd nodes are 21,23,26,27, and the even nodes are 21, 25, 24. Note that we are not bothered about the value of the nodes. We append the even nodes list at the end of the odd node list.

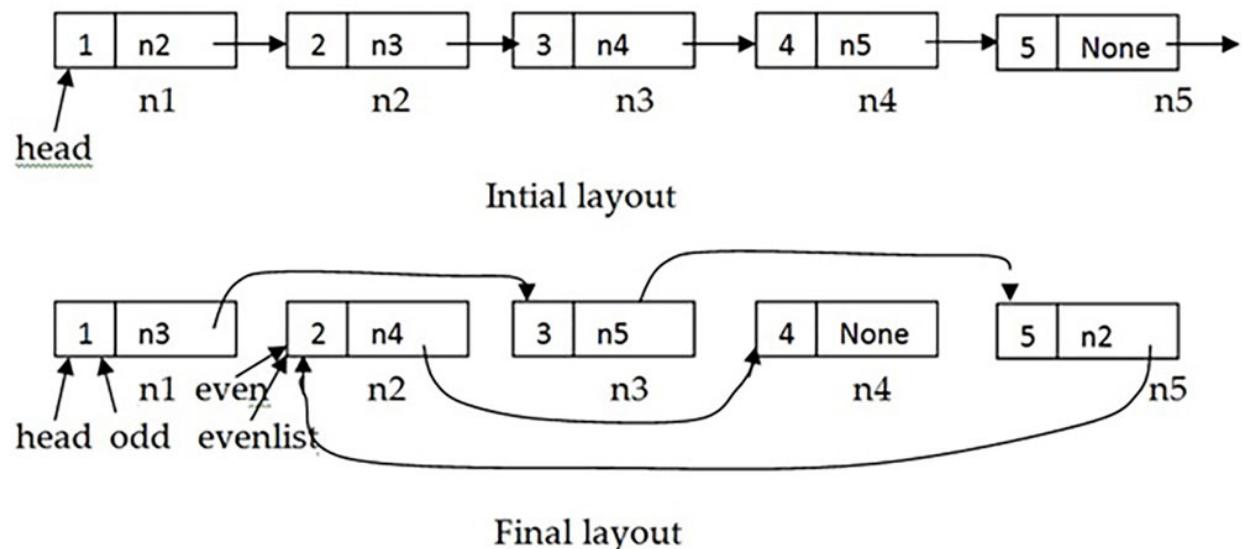
## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def oddEvenList(self, head: ListNode) -> ListNode:
```

## Strategy

Consider the layout of the list 1->2->3->4->5 shown in [Figure 2.5](#):



*Figure 2.5: Initial and final layout*

Initially, the head points to *n1*. We will make the next pointer jump two steps at a time to mop up the odd numbered elements. We will use a variable *odd* to iterate over **odd** numbered elements. Initially, *odd* = *head*. The element to the right of the *head* will be the start of the even list. We will iterate the even list with the variable *even*. Initially, *even* = *evenlist*. When the odd pointer

reaches the end of the list, we will make the next pointer of the last element equal to the *evenlist*. Thus, the even list will get attached at the end.

In each iteration, we will perform the following steps as long as the element and *even.next* exist. (Assume the pointer positions as shown above.)

Initially, the variables are:

odd = n1, even = n2, **list** = [1:n2, 2:n3, 3:n4, 4:n5, 5:None]

1. *n1.next* should point to *n3*, which is *n2.next* now, that is, *odd.next* = *even.next*.

Variables: odd = n1, even = n2, **list** = [1:n3, 2:n3, 3:n4, 4:n5, 5:None]

2. The pointer odd should advance from *n1* to *n3*, that is, *odd* = *odd.next*.

Variables: odd = n3, even = n2, **list** = [1:n3, 2:n3, 3:n4, 4:n5, 5:None]

3. *n2.next* should point to *n4*, which is *n3.next* now, that is, *even.next* = *odd.next*.

Variables: odd = n3, even = n2, **list** = [1:n3, 2:n4, 3:n4, 4:n5, 5:None]

4. The pointer even should advance to *n4*, that is, *even* = *even.next*

Variables: odd = n3, even = n4, **list** = [1:n3, 2:n4, 3:n4, 4:n5, 5:None]

When the loop is over, we have two chains. The odd chain starting from the head, and the even chain starting from the *evenlist*. We will finally append the even chain at the end of the odd chain.

## Python code

The following code implements this strategy:

```
1. def display(root):  
2.     while (root != None) :  
3.         print(root.val, end = " ")  
4.         root = root.next  
5.     print('\n')  
6. class ListNode:  
7.     def __init__(self, val=0, next=None):  
8.         self.val = val  
9.         self.next = next
```

```

10. class Solution:
11.     def oddEvenList(self, head: ListNode) -> ListNode:
12.         if(not head):    #If head is invalid, return
immediately
13.         return head
14.
15.         odd = head           #odd = iterator of odd
series
16.         even = odd.next       #even = iterator of even
series
17.         evenList = even        #Start of even series
18.
19.         while(even and even.next):
20.             odd.next = even.next
21.             odd = odd.next
22.
23.             even.next = odd.next
24.             even = even.next
25.
26.         odd.next = evenList   #Append start of even series to
end of odd
27.         return head
28. #Driver code
29. #Create list 11,12,13,14,15
30.
n1=ListNode(11);n2=ListNode(12);n3=ListNode(13);n4=ListNode(14);
n5=ListNode(15)
31. n1.next = n2;n2.next = n3;n3.next = n4;n4.next=n5;n5.next =
None
32. display(n1)
33. sol=Solution()
34. r=sol.oddEvenList(n1)
35. display(r)

```

## **Output:**

11 12 13 14 15

11 13 15 12 14

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The whole list is traversed. So, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are not using any extra space, the space complexity is  $O(1)$ .

## 2.8 Question 21 – How will you evaluate the reverse polish postfix expression?

Normally, we write the arithmetic expressions in everyday life and in programming languages using the BODMAS rule. This is also called the infix notation because the arithmetic operator is fixed between the operands.

For example:

$1 + 2 * 3$  is evaluated as  $1 + (2*3) = 1 + 6 = 7$

The compiler converts it to the reverse polish notation because it is easy to evaluate. It is also called the polish postfix notation. In this notation, the operands are written one after the other and the operator is written after that. The above expression in the reverse polish notation is written as:

1 2 3 \* +

## Problem statement

Evaluate a string representing an arithmetic expression written in the reverse polish notation. Note: Division must be the integer division. Let us illustrate with the help of two examples.

### **Example 1:**

**Input:** [“2”, “5”, “+”, “3”, “\*”]

**Output:** 9

**Explanation:** Elements 2 and 5 are stored in a stack. Then + operator is found. The two stack top elements are added and the result is pushed on the stack. Element 3 is read and pushed on the stack. The operator \* is found. The two stack top elements are multiplied to give the result. This is equivalent to the expression  $((2 + 5) * 3) = 21$ .

### Example 2:

**Input:** ["1", "13", "5", "/", "+"]

**Output:** 6

**Explanation:** Following the similar logic explained in the previous example, the input is equivalent to the expression  $(1 + (13 / 5)) = 3$

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def evalRPN(self, tokens: List[str]) -> int:
```

### Strategy

The RPN expression consists of a list of tokens, that is, operands or operators. The procedure for evaluation is as follows:

Read a token from the list of tokens. If it is an operand, push it on a stack. If it is an operator, pop two operands from the stack and perform the operation on them. The result is pushed back on the stack. When the tokens are finished, the answer is on the stack top.

**For evaluation of operators, let me introduce the use of lambda functions. We could have achieved the same result with an if-else-if ladder, but that would be slower.**

Lambda functions are convenient single line function definitions. For example, let us define a function avg that returns the average of its two arguments:

```
avg = lambda a, b: (a + b)/2
```

We can call the function like this:

```
print(avg(6, 7))
```

## Output: 6.5

In our example, we want to call the evaluation function corresponding to the operator. We will, therefore, enter the operators and their lambda functions in a dictionary op. The topic of the dictionary is discussed in detail in the next chapter. For now, it is sufficient to know that a dictionary is a set of key-value pairs. In our case, the ops dictionary will contain the operator-function pairs. For example:

```
“+”: (lambda a, b: a + b)
```

The problem states that the integer arithmetic must be used. This creates anomaly if // operator is used. The // operator intrinsically applies floor function to the result of division. It produces an unexpected result in the integer division operation if one of the operands is negative. Consider:

$2//5 = \text{floor}(.4) = 0$

$2//(-5) = \text{floor}(-.4) = -1$  (Integer smaller than -.4 is -1)

$\text{int}(2/(-5)) = \text{int}(-.4) = 0$  (int removes fraction. This is the intended operation)

Therefore, we should use the normal division and convert it into integer.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. #define lambda functions in a dictionary
3. ops = {
4.     “+”: (lambda a, b: a + b),
5.     “-”: (lambda a, b: a - b),
6.     “*”: (lambda a, b: a * b),
7.     “/”: (lambda a, b: int(a / b)) #See above discussion on
integer arithmetic
8. }
9. class Solution:
```

```

10. def evalRPN(self, expression: List[str]) -> int:
11.     stack = []
12.     for token in expression: #Examine each token
13.         if token in ops:           #If token is an operator
14.             arg2 = stack.pop()    #Then pop two elements from the
stack
15.             arg1 = stack.pop()
16.             result = ops[token](arg1, arg2)#Perform the operation
17.             stack.append(result)        #Push the result on
stack
18.         else:
19.             stack.append(int(token))   #If token is not
operator, push it on stack
20.
21.     return stack.pop()      # In the end, stack top contains
the answer
22. #Driver code
23. sol = Solution()
24. print(sol.evalRPN(["2", "5", "+", "3", "*"]))
25. print(sol.evalRPN(["4", "13", "5", "/", "+"]))
26. print(sol.evalRPN(["10", "6", "9", "3", "+", "-11", "*",
"/", "*", "17", "+", "5", "+"]))

```

## **Output:**

```

21
6
22

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

If  $n$  is the length of the input list, the *for* loop runs  $n$  times. Thus, the time complexity is  $O(n)$ .

## **Space complexity:**

As we are using stack, the space complexity is  $O(n)$ .

## **2.9 Question 22 – How will you achieve the minimum implementation of a stack using a list?**

The list class defined in Python is actually a dynamic array. It already has a built-in functionality of a stack (push and pop). However, in this implementation, we want one more function to get the minimum element, that too with  $O(1)$  time complexity.

### **Problem statement**

Design the minimum implementation of a stack that supports the following operations in constant time:

- $push(x)$  – Pushes the element  $x$  onto the stack.
- $pop()$  -- Removes the element from the top of the stack.
- $top()$  -- Gets the top element without removing it.
- $getMin()$  -- Retrieves the minimum element in the stack.

### **Solution format**

Your solution should be in the following format:

```
1. class MinStack:  
2.     def __init__(self):  
3.     def ...  
4.     def ...
```

### **Strategy**

We will study two approaches. The first approach is simple. It uses a search technique to get the minimum. However, it does not satisfy the requirement of the  $O(1)$  time complexity stated in the problem. For getting  $O(1)$ , we will use the second approach.

## Approach 1: O(n)

Python List class has all the functionalities of the stack built-in, except the *getMin* function. We will create an empty array, named *stk* during the initialization. *Push* is implemented as *append* and *pop* is implemented as *pop()*. *Top* is equivalent to addressing *stk[-1]*. It is very tempting to ignore the constant time specification for *getMin* in the first attempt. Let us try this approach first. We will initialize a variable *min* to a very large value. Then, we will scan the array and at each iteration, update *min* with the minimum of *min* and *stk[n]*.

## Python code

The following code implements this strategy:

```
1. class MinStack:  
2.     def __init__(self):  
3.         self.stk = []          #Create empty stack  
4.     def push(self, x: int) -> None:  
5.         self.stk.append(x)  
6.     def pop(self) -> None:  
7.         self.stk.pop()  
8.     def top(self) -> int:  
9.         return self.stk[-1]  
10.    def getMin(self) -> int:  
11.        mini = float('inf')  
12.        for n in self.stk:  
13.            mini=min(mini,n)  
14.        return mini  
15.  
16. #Driver code  
17. obj = MinStack()  
18. obj.push(-2)  
19. obj.push(0)  
20. obj.push(-3)  
21. print(obj.stk)
```

```
22. print(obj.getMin())
23. obj.pop()
24. print(obj.stk)
25. print(obj.getMin())
```

### **Output:**

```
[-2, 0, -3]
-3
[-2, 0]
-2
```

### **Approach 2 – O(1)**

The *getMin* function in approach 1 has a time complexity of  $O(n)$  and the space complexity of  $O(1)$ . We can achieve the  $O(1)$  time complexity by trading the space complexity. We will create a class variable *mini* to store the minimum value. When a new member is added with the *push* operation, we will compare it with *mini* and update *mini*, if required. The *getMin* function simply returns *mini*.

But the problem arises in the *pop* operation if the element being popped is equal to *mini*. We don't have any information on what is the next minimum. We will store this information in the same stack but taking care that *top* and *pop* operations function the same way. During the *push* operation, if we find that the new element is less than the *mini*, we will first push the existing *mini* value, update *mini*, and then push the new element. During the *pop* operation, if we find that the element being popped is equal to *mini*, we will pop one more element from the *stack* and store it in *mini*.

### **Python code**

The following code implements this strategy:

```
1. class MinStack:
2.     def __init__(self):
3.         self.stk = []                      #Create empty stack
4.         self.mini = float('inf')
5.     def push(self, x: int) -> None:
```

```

6.         if x<=self.mini:           #If new value is less than
mini
7.             self.stk.append(self.mini) #Then save old val of
mini on stack
8.             self.mini = x          #Update mini
9.             self.stk.append(x)      #Append new element
10.            def pop(self) -> None:
11.                stktop=self.stk[-1]
12.                self.stk.pop()
13.                if self.mini == stktop:
14.                    self.mini = self.stk[-1]
15.                    self.stk.pop()
16.
17.            def top(self) -> int:
18.                return self.stk[-1]
19.
20.            def getMin(self) -> int:
21.                return self.mini

```

## **Output:**

```

[-2, 0, -3]
-3
[-2, 0]
-2

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

The time complexity for *push()*, *pop()*, and *top()* is  $O(1)$  in the first approach. These operations take more time in the second approach, but the time is independent of  $n$ . So, the time complexity is still  $O(1)$ . The *getMin* function in approach 1 has to scan the array, so its time complexity is  $O(n)$ . In the second approach, the minimum element can be given without searching, so its time complexity is  $O(1)$ .

### **Space complexity:**

For the first approach, no extra space is required. So, the space complexity is  $O(1)$ . For the second approach,  $n$  extra elements are required to store the minimum values. So, the space complexity is  $O(n)$ .

## **Conclusion**

We started with an explanation of the basic operations on the linked lists. We studied 8 problems that sharpened your skills in manipulating the linked lists. We studied the binary search technique applied to various situations. We studied how to trade space complexity with time complexity.

In the next chapter, we will study the questions based on hash tables and general Math.

## **Points to Remember**

- Linked list is made up of nodes.
- A node is a data structure consisting of value and pointer to the next node.
- The first element of a list is called the *head*, and the last element is called the *tail*.
- The next pointer of the tail is *None* if the list has no cycles.
- We use the keyword *None* to represent null objects.
- Lambda functions are single-line function definitions.
- The push and pop operations are inherently included in the list class.
- Linked list is useful if frequent insertion and deletion is required.

## **MCQs**

1. In the worst case, what is the number of comparisons needed to search a singly linked list of length  $n$  for a given element?

- A.  $\log_2 n$
- B.  $n/2$

- C.  $\log_2 n - 1$
  - D. n
2. It takes 1 microsecond to insert a new node at the beginning of a list that already has 100 nodes. How many microseconds will be required if the list has initially 200 nodes?
- A. 1
  - B. 100
  - C. 200
  - D. 400
3. It takes 100 microseconds to insert a new node at the end of a list that already has 100 nodes. How many microseconds will be required if the list has initially 200 nodes?
- A. 10
  - B. 100
  - C. 200
  - D. 400
4. There is a type of linked list in which the last node of the list points to the first node of the list. What is it called?
- A. Singly linked list
  - B. Doubly linked list
  - C. Circular linked list
  - D. Multiply linked list
5. What is the time complexity of reversing a linked list in-place?
- A. O(1)
  - B. O(n)
  - C. O( $\log n$ )
  - D. O( $2^n$ )

## Answers to MCQs

**Q1:** D

**Q2:** A

**Q3:** C

**Q4:** C

**Q5:** B

## **Questions**

1. What are the constituents of the linked list node?
2. Can we do a binary search on a linked list?
3. Name some applications of linked lists.
4. Explain the terms head and tail of a linked list?
5. What is a circular linked list?

## **Key terms**

Node, linked list, head, tail, circular linked list, cycle, traversing, reverse polish notation, lambda functions.

## CHAPTER 3

### Hash Table and Maths

The hash table is a special kind of array where an item is not accessed by its index, but by searching the data in the array. You might ask if we already know the data to be searched, why access the array? Well, we don't know the complete data. An element stored in the hash table consists of two parts, namely key and value. The key may be an integer or a string. For accessing the data, we search the key, and when found, retrieve the corresponding value. Such type of storage is also called "content addressable memory" or "associative memory" because it mimics human memory. When we append an element to a hash table, a "hashing function" generates an index by doing some arithmetic operations on the key. The element is stored at that index. When we want to retrieve the element, we don't have to search the array. We use the same hashing function on the key and get the index directly. Thus, the time complexity of the search operation is  $O(1)$ . This is better than a simple array which gives the time complexity of  $O(n)$  with the linear search and  $O(\log n)$  with the binary search.

### Structure

We will cover the topics in the following order:

- Implementation of a hash table in Python
- Question 23: How will you find if an array contains duplicates?
- Question 24: How will you find if an array contains duplicates in the vicinity?
- Question 25: How will you find the majority element in an array?
- Question 26: How will you find if the string of brackets is valid?
- Question 27: How will you find two numbers whose sum equals the target?
- Question 27: How will you count the primes less than  $n$ ?

- Question 29: How will you find the longest substring without the repeating characters?
- Question 30: How will you convert a roman numeral into a decimal numeral?
- Question 31: How will you identify a single number in an array?

## Objectives

After studying this unit, you should be able to:

- Understand the creation and manipulation of hash tables
- Learn to use hash tables to implement fast algorithms
- Get mastery over math operations

### 3.1 Implementation of Hash Table in Python

The hash table is implemented in Python as a dictionary. But there is no keyword called “dictionary” in Python. When the key-value pairs are enclosed in curly brackets, they are recognized as a dictionary. Similarly, the items enclosed in square brackets are recognized as a “list”, and the items enclosed in round brackets are recognized as a “tuple”. As an example, let us create a dictionary of persons’ names and their ages. You can type the commands in the Python command prompt.

```
Dict = {"Sam":70,"Leena":23,"John":32}
```

For a user, the dictionary appears just like an array, except that the index need not be an integer; it can also be a string. The key acts like an index into the array. If we want to print the age of Leena, we will use:

```
print(Dict["Leena"])
```

The output will be 23.

A new entry George:22 can be added like this:

```
Dict["George"] = 22
```

To test whether a given key exists in the dictionary or not, use the `in` command:

```
print('Leena' in Dict) prints True.
```

print('Meena' in Dict) prints False because Meena does not exist in the dictionary.

Now type this command.

```
print(Dict["Meena"])
```

What will happen?

The console prints:

```
Key error: "meena"
```

But if the above command is being executed from a script, the system generates an exception and the program gets terminated. If you want to avoid this, it is better to use a safer method “get”:

```
print(Dict.get("Meena")) prints None. The program does not get terminated.
```

The get function takes in a second argument which defaults to None. If the second argument is given, the function returns that argument if the key is not found in the dictionary. For example, consider the following commands:

```
print(Dict.get('Leena', 0))
```

The output will be 23.

```
print(Dict.get('Meena', 0))
```

The output will be 0.

## Using “defaultdict”

There is a convenient version of the dictionary called the defaultdict, which is derived from the standard dictionary class. It is defined in the module collections. Therefore, if you want to use it, you must include this line in the beginning of your program:

```
from collections import defaultdict
```

It has all the functionalities of the standard dictionary and in addition, it handles the non-existent keys conveniently. There is no need to use the “get” method. You just access the content by giving the key in square brackets. If the key is non-existent, it does not give an error but returns a default value. The defaultdict cannot be automatically created using curly brackets. We need to call the defaultdict function as follows:

```
dd = defaultdict()
```

With the above call, the default value is set to None. We can set the default value to 0 by the following command:

```
dd = defaultdict(int)
```

We can set the default value to an empty list by the following command:

```
dd = defaultdict(list)
```

## Using “set”

In Python, the set can be considered as a dictionary that has only keys and no values. The keys must not be duplicates; they must be unique. It is created by enclosing the items in curly brackets. For example:

```
Flowers = {"Rose", "Lily", "Lavender"}
```

However, an empty set cannot be created by the command {}. It creates a dictionary. An empty set can be created like this:

```
Flowers = set()
```

You can add elements as follows:

```
Flowers.add("Rose")
```

You can use a keyword to check the presence of an item in the set. For example:

“Rose” in flowers prints True.

The set is used for membership testing, removing duplicates, and set operations like union and intersection.

## 3.2 Question 23 – How will you find if an array contains duplicates?

Several approaches for this have been suggested so that you get mastery over the language and learn to discover the advantages and disadvantages of various approaches. The use of set and dictionary is explained.

### Problem statement

Find if the given array contains any duplicates. If any value appears two or more times, your function should return `true`, and if every element is distinct, it should return `false`. Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:** [5, 2, 3, 5]

**Output:** True

**Explanation:** The number 5 is duplicate, so, the output is True.

#### **Example 2:**

**Input:** [5, 6, 7, 8]

**Output:** False

**Explanation:** All the numbers are distinct, so, the output is False.

#### **Example 3:**

**Input:** [7, 7, 7, 3, 3, 4, 3, 2, 4, 2]

**Output:** True

**Explanation:** The numbers 2,3,4,7 have duplicates, so, the output is True.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def containsDuplicate(self, nums: List[int]) -> bool
```

### Strategy

We will study four approaches so that you can get a good grip over the language and get to know the advantages and disadvantages of various approaches.

## Approach 1 – Brute force

A simple but time-consuming method is to identify out all pairs `nums[i]` and `nums[j]` and test if they are equal.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def containsDuplicate(self, nums):  
3.         for i in range(len(nums)):  
4.             for j in range(i+1, len(nums)):  
5.                 if nums[i] == nums[j]:  
6.                     return True  
7.         return False  
8. #Driver code  
9. sol=Solution()  
10. print(sol.containsDuplicate([1,2,3,4]))  
11. print(sol.containsDuplicate([1,1,1,3,3,4,3,2,4,2]))
```

## **Output:**

False

True

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Due to the two nested loops, the time complexity is  $O(n^2)$ .

### **Space complexity:**

As we are not using extra space, the space complexity is  $O(1)$ .

## Approach 2 – Sorting

We will sort the array using the built-in sort method and check if any consecutive elements are identical.

### Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def containsDuplicate(self, nums):  
3.         nums.sort()  
4.         for i in range(len(nums)-1):  
5.             if nums[i] == nums[i+1]:return True  
6.         return False
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Sorting has the time complexity of  $O(n \log n)$ . After that, we have one cycle of length  $n$ . Hence, the time complexity is  $O(n \log n + n) = O(n \log n)$ .

### **Space complexity:**

As we are not using extra space, the space complexity is  $O(1)$ .

## Approach 3 – Using the “set” method

The previous two approaches are suitable for the C language. Python has powerful built-in classes, namely set and dictionary. They make the program concise and fast. Here, we will use the built-in set() method. It converts a list (or any other iterable collection) into a sequence of iterable elements with distinct elements, that is, the duplicates are removed. In addition, the elements are sorted. For example, type the following in the command prompt.

```
set([3,4,1,4,5])
```

Output is {1, 3, 4, 5}.

To detect the duplicates, we will compare the size of the original array and the size of the set. If there are duplicates, then the size of the set will be smaller. This logic can be implemented in a single line.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def containsDuplicate(self, nums):  
3.         return len(set(nums)) < len(nums)
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The set insertion has the time complexity of  $O(1)$ . Inserting  $n$  elements will be  $O(n)$ . The length operation is  $O(1)$  in both the sets and lists. Thus, the overall time complexity is  $O(n)$ .

### **Space complexity:**

As we are creating a set, the space complexity is  $O(n)$ .

## Approach 4: Using hash tables

Here, we will use a hash table, instead of a set. First, we will create an empty hash table. Then, we will scan the `nums` array in a loop and in each iteration, we will examine if the number from the `nums` array is present in the hash table. If yes, then we have found a duplicate, so return `true`. Otherwise, at the end of the loop, return `false`.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def containsDuplicate(self, nums):  
3.         htable = {}  
4.         for n in nums:
```

```
5.         if n in htable: return True
6.         else: htable[n] = 1
7.     return False
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The hash table has a time complexity of  $O(1)$  for insertion. We are running the loop  $n$  times. Hence, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are creating a hash table, the space complexity is  $O(n)$ .

## 3.3 Question 24 – How will you find if an array contains duplicates in the vicinity?

This is an extension of the previous question.

### Problem statement

Find if the given array contains any duplicates that are separated by a distance, no greater than  $k$ .

This is similar to the last problem with an added constraint that duplicates are recognized only if they are within a distance of  $k$ , that is, if  $\text{nums}[i]=\text{nums}[j]$ , and the absolute difference between  $i$  and  $j$  is, at the most,  $k$ . Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:**  $\text{nums} = [1, 4, 3, 1]$ ,  $k = 3$

**Output:** True

**Explanation:** 1 occurs at 0 and 3. The distance (3-0) is within the limit of 3. Hence, they are duplicates in the vicinity, and the output is True.

#### **Example 2:**

**Input:**  $\text{nums} = [1, 5, 1, 1]$ ,  $k = 1$

**Output:** True

**Explanation:** 1 occurs at 0, 2, and 3. The distance (3-2) is within the limit 3. Hence, they are duplicates in the vicinity, and the output is True.

### Example 3:

**Input:** `nums = [1, 2, 3, 1, 2, 3], k = 2`

**Output:** False

**Explanation:** 1 occurs at 0 and 3, 2 occurs at 1 and 4, and 3 occurs at 2 and 5. All distances (3-0), (4-1), and (5-2) exceed the limit 2. Hence, there are no duplicates in the vicinity, and the output is false.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
```

## Strategy

We will first create an empty hash table `dict`. It will store the pairs `(nums[i]:i)`. If `nums[i]` is repeated, the hash table will contain the index of its latest occurrence. Then, we will scan the `nums` array. If the element being scanned i.e. `nums[i]` has been previously entered in the `dict`, we will check if its index differs from `i` by less than `k`. If yes, return `True`. Otherwise, continue scanning. In the end, return `False`.

## Python code

The following code implements this strategy:

```
1. from typing import List  
2. class Solution:  
3.     def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:  
4.         htable = {}  
5.         for i in range(len(nums)):  
6.             num = nums[i]
```

```

7.         if num in htable and i - htable[num] <= k:
8.             return True
9.         htable[num] = i
10.        return False
11. #Driver code
12. sol=Solution()
13. print(sol.containsNearbyDuplicate([1,2,3,1],3))
14. print(sol.containsNearbyDuplicate([1,2,3,1,2,3],2))

```

### **Output:**

True

False

### **Complexity Analysis**

The complexity analysis is as follows:

#### **Time complexity:**

The hash table has a time complexity of  $O(1)$  for insertion and membership check. Hence, the overall time complexity is  $O(n)$ .

#### **Space complexity:**

As we are creating a hash table with  $n$  entries, the space complexity is  $O(n)$ .

### **3.4 Question 25 – How will you find the majority element in an array?**

This question introduces the use of a hash table for counting.

#### **Problem statement**

Given an array of size  $n$ , find the majority element, that is, the element that appears the maximum number of times. You may assume that the array is not empty. Let us illustrate this with the help of two examples.

#### **Example 1:**

**Input:** [3,2,3]

**Output:** 3

**Explanation:**  $n=3$ . Element 3 occurs the most (2 times).

**Example 2:**

**Input:** [2,2,1,1,3,1,2,2]

**Output:** 2

**Explanation:** Element 2 occurs the most (4 times).

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def majorityElement(self, nums: List[int]) -> int:
```

## Strategy

We will create a dictionary, named `counts` which will contain the occurrence count of each value in the array. There are several approaches to do this. We will start from a simple intuitive approach and gradually proceed to approaches that exploit the special features of Python.

## Approach 1 – From basics

We will create an empty dictionary, named `counts`. It will contain the pairs, `value:count`. Then we will scan the `nums` array. If we find `nums[i]` in `counts`, we will increment its count, otherwise, we will make a new entry (`nums[i]:count`). Finally, we will use the built-in method `max` to get the key corresponding to the maximum value. The second parameter of the `max` function, that is, `key=counts.get` specifies that the maximum of the values is to be computed.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def majorityElement(self, nums):  
3.         counts = {}  
4.         for item in nums:
```

```

5.         if (item in counts):
6.             counts[item] += 1
7.         else:
8.             counts[item] = 1
9.     return max(counts, key=counts.get)
10. #driver code
11. sol = Solution()
12. print(sol.majorityElement([2,2,1,3,1,1,2,2]))

```

### **Output:**

2

### **Approach 2 – Using the “count” method of the list class**

The Python list class has a built-in method “count” which returns the number of times an argument occurs in the list. We will scan the `nums` array with the iterator `items`. The expression `nums.count(items)` evaluates to the count of the occurrences of `item` in `nums`. We will enter the pair (`items:count`) in the dictionary, named `counts`. In the end, we will get the maximum by the same method we used earlier.

### **Python code**

The following code implements this strategy. The driver code from the previous solution can be reused. The output is the same.

```

1. class Solution:
2.     def majorityElement(self, nums):
3.         counts = {}
4.         for items in nums:
5.             counts[items] = nums.count(items)
6.         return max(counts, key=counts.get)

```

### **Approach 3 – Using the “get” method of the dictionary with default**

Let us modify approach 1. We had to test if the dictionary contained the item, because if it did not exist and we tried to access it, the program would get terminated. Let us use the safer “get” method. As we know, it takes an optional parameter that specifies the default value to be returned when the search key is not found. By default, the default value is `None`. But we will specify it to be 0 in the second parameter. Now, we can straightforwardly increment the `counts[item]`. If the key item is absent, the `get` method returns 0. You can also try the `defaultdict` class from the `collections` module instead of the standard dictionary.

In the end, we will get the maximum by the same method we used earlier.

## Python code

The following code implements this strategy. The driver code from the previous solution can be reused. The output is the same.

```
1. class Solution:  
2.     def majorityElement(self, nums):  
3.         counts = {}  
4.         for item in nums:  
5.             counts[item] = counts.get(item, 0) + 1  
6.         return max(counts.keys(), key=counts.get)
```

## Approach 4 – Using the “Counter” method from the collections module

We will use the `Counter` method from the `collections` module. When we give an array as an input to the `Counter`, it returns a dictionary containing the element values, and their occurrence counts in the form of `(value:count)`. The usage will be clear from the following example. Type the following commands in the Python command prompt.

```
import collections  
collections.Counter([1,1,2,2,2,2,3,4,4])
```

The output is:

```
Counter({2: 4, 1: 2, 4: 2, 3: 1})
```

From the output, we know that 1 occurs twice, 2 occurs 4 times, 3 occurs once, and 4 occurs twice.

We will collect the counts and get the maximum by the same method we used earlier.

## Python code

The following code implements this strategy. The driver code from the previous solution can be reused. The output is the same.

```
1. import collections
2. class Solution:
3.     def majorityElement(self, nums):
4.         counts = collections.Counter(nums)
5.         return max(counts.keys(), key=counts.get)
```

## Complexity Analysis

The complexity analysis of the last approach is as follows:

### **Time complexity:**

The creation of counts with the Counter function proceeds with the time complexity of  $O(n)$ . Finding the maximum also proceeds with the time complexity of  $O(n)$ . The other operations are  $O(1)$ . Hence, the overall time complexity is  $O(n)$ .

### **Space complexity:**

As we don't need additional space, the space complexity is  $O(1)$ .

## 3.5 Question 26 – How will you find if the string of brackets is valid?

This question uses hash tables for the fast processing of the strings.

## Problem statement

You need to analyze a string consisting of only brackets, that is, '(', ')', '{', '}', and '[', ']' to determine if it is valid.

It is valid if:

1. The opening brackets are closed by the same type of closing brackets.
2. The opening brackets are closed in the correct order.

It is worth noting that an empty string is also considered valid.

Let us illustrate this with the help of some examples.

**Example 1:**

**Input:** “()”

**Output:** True

**Explanation:** The opening round bracket is balanced by the closing round bracket. Hence, the output is True.

**Example 2:**

**Input:** “()[]{}”

**Output:** True

**Explanation:** Each opening bracket is balanced by a closing bracket of the same type. Hence, the output is True.

**Example 3:**

**Input:** “{}”

**Output:** False

**Explanation:** The opening bracket does not match the closing bracket. Hence, the output is False.

**Example 4:**

**Input:** “({})”

**Output:** False

**Explanation:** The opening and closing brackets do not match. Hence, the output is False.

**Input:** “({})”

**Output:** True

**Explanation:** The brackets are nested and they are in the correct order. Hence, the output is True.

**Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def isValid(self, s: str) -> bool:
```

## Strategy

The question of matching a bracket arises only when we encounter a closing bracket. So, as long as we encounter an opening bracket, we will simply push it on the stack. When we encounter a closing bracket, we will check if the stack top contains the matching opening bracket. For quickly searching the correct opening bracket that pairs with the closing bracket, we will use a dictionary, named `brac_match`. To determine whether the given character is one of the openers, we will use another dictionary, named `openers`.

## Pseudo code

Scan the given string, one character at a time.  
For each character, check:  
If the character is an opening parenthesis of any type:  
Push it on the stack.  
Else if stack is empty:  
 There exist unbalanced closers, return False.  
Else if character matches with character last pushed on stack –  
 '()' for ')', '[]' for ']', '{}for '}':  
 OK. Pop the character from the stack and move on.  
Else (i.e. char is not opener, stack is not empty, char does not match stack top opener)  
 Return False.

When the string is done, the stack should be empty. If it is empty, return True, else return False.

## Python code

The following code implements this strategy:

```
1. class Solution:
```

```

2.     def isValid(self, s: str) -> bool:
3.         brac_match = {')': '(', ']': '[', '}': '{'}
4.         openers = ['(', '[', '{']
5.         stk = []#Define array to act as a stack
6.         for char in s:
7.             if char in openers:
8.                 stk.append(char)#If char is opening bracket,
push it on stack
9.             elif [] == stk:
10.                 return False #if stack is empty when closing
comes, it is False #condition
11.             elif stk[-1] == brac_match[char]:
12.                 stk.pop()
13.             else:
14.                 return False
15.             return [] == stk #If you end up in empty stack,
return True
16. #Driver code
17. sol = Solution() #Instantiate an object of Solution class
18. print(sol.isValid("[]")) #Call member function isValid.

```

**Output:** True

## Complexity Analysis

### **Time complexity:**

We observe that there is a main loop in the program for the char in s. If the length of s is  $n$ , the loop will run  $n$  times. So, the time complexity is  $O(n)$ .

### **Space complexity:**

The extra memory space needed is for the stack. We don't know its exact length, but it is expected to be proportional to  $n$ . So, the space complexity is  $O(n)$ .

## 3.6 Question 27 – How will you find two numbers whose sum equals the target?

This is a math-oriented problem that can be made fast by applying hash tables.

## **Problem statement**

You are given an array of integers, `nums`, and a number, `target`. If any two numbers in `nums` add up to the specified target, then return their indices. You may assume that each input would have exactly one solution, and you may not use the same element twice. Let us illustrate this with the help of an example.

### **Example:**

Given `nums` = [3, 8, 13, 15], `target` = 11

**Output:** [0, 1]

**Explanation:** `nums[0] + nums[1] = 3 + 8 = 11`. So, we return the indices 0 and 1.

## **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def twoSum(self, nums: List[int], target: int) ->  
List[int]:
```

## **Strategy**

We will try two approaches, namely the brute force based and the hash table based.

## **Approach 1 – Brute force method**

This is slow but if it works, it gives you enough credits in your interview. This gives you an intuition to look for improved solutions. Let us find all the possible pairs of numbers and find whether their sum is equal to the target. This could be done by running two nested loops as follows.

## **Python code**

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def twoSum(self, nums: List[int], target: int) ->
List[int]:
4.         for i in range(len(nums)):
5.             for j in range(i+1, len(nums)):
6.                 if nums[i] + nums[j] == target:
7.                     return [i, j]           # Return array [i,j]
8. #Driver code
9. sol = Solution()
10. print(sol.twoSum([2, 7, 11, 15],9))
```

**Output:** [0,1]

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Since there are two nested loops, the time complexity is  $O(n^2)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## Approach 2 – Hash table

The above strategy gave us the time complexity of  $O(n^2)$ . Can we do better? We observe that for every value of  $i$ , the inner loop searches the elements of the `nums` array to find an element that adds up to the target. This makes the time complexity of the inner loop  $O(n)$ . We can reduce it to  $O(1)$  with the following strategy.

Consider that every element `nums[i]` has a complement, which is  $(\text{target} - \text{nums}[i])$ . For example, if the target is 9, then the complement of 2 is 7, the complement of 7 is 2, the complement of 11 is -2, and so on. What the inner loop of the first approach is doing is searching the complement of the number. For quickly searching it, we will build a dictionary (hash table) with

the number as the key and the corresponding index as the value. In the above example, the hash table would look like this.

```
{2:0, 7:1, 11:2, 15:3}
```

We will first create an empty dictionary, named `htable`. Then we will scan the `nums` array indexed by `i`. If the complement of `nums[i]` exists in the dictionary, we have found our solution. We will return the index of the complement and `i`.

If the complement does not exist in the dictionary, we will add the pair `number:i` to the dictionary with the statement `htable[nums[i]]=i`, hoping that in the future, we will find a number whose complement is this number.

Let us walk through the example. Initially, we will create an empty hash table, `htable`. Then we will start the loop. In the first iteration, `i=0` and `nums[i] = 2`. We will not find the complement of 2, that is, 7 in `htable` as it is empty. So, we will add the entry `2:0` to `htable`.

In the next iteration, `i=1` and `nums[i] = 7`. Now, the `htable` does not disappoint us as the complement of 7, that is, 2, is present there and we have found our solution.

If the whole string is scanned and we don't find a complement, the function returns without a return value. Note that this path is not visible in the code. The function simply returns when its job is done. The `print` statement prints `None` in such cases.

A C compiler shouts foul if different return paths give different arguments but with Python, it is OK.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def twoSum(self, nums: List[int], target: int) ->
List[int]:
3.         htable = {}
4.         for i in range(len(nums)):
```

```

5.             if target - nums[i] in htable: #Search for
complement of nums[i]
6.                 return [htable[target - nums[i]],i] #return
locof compl and i
7.             else
8.                 htable[nums[i]]=i #Enter the number and its
location in dictionary
9. sol = Solution()
10. print(sol.twoSum([2, 7, 11, 15],9))

```

**Output:** [0,1]

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The time complexity of the look-up in the dictionary is  $O(1)$ . Hence, the time complexity of the inner loop is  $O(1)$ . We managed to reduce it from  $O(n)$  to  $O(1)$ . The time complexity of the algorithm is decided by the outer loop. It is  $O(n)$ .

### **Space complexity:**

We have created a dictionary of length  $n$ . So, the space complexity is  $O(n)$ . Thus, we have traded time for space.

## 3.7 Question 28 – How will you count the primes less than n?

This is a math-oriented problem where we can build a better solution if we know some Math algorithms. Let us illustrate this with the help of an example.

### Problem statement

Count the number of prime numbers less than a given number  $n$ . Assume that  $n$  is non-negative.

### **Example:**

**Input:** 12

**Output:** 5

**Explanation:** There are 5 prime numbers less than 12 namely 2, 3, 5, 7, and 11.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def countPrimes(self, n: int) -> int:
```

## Strategy

We will study this with two approaches. The first is based on brute force and the second is based on the Eratosthenes sieve methodology.

## Approach 1 – Brute force method

We will write a helper function `isPrime(n)` which returns `True` if  $n$  is prime. In the main function, we will first initialize the count to zero, and then start a loop where  $i$  goes between 2 and  $n$ . We will call the helper function `isPrime` to test if  $i$  is prime and whenever  $i$  is found to be prime, the count is incremented.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def countPrimes(self, n: int) -> int:
3.         if n<2:                                #Base cases
4.             return 0
5.         count = 0
6.         for i in range(2,n):
7.             if self.isPrime(i):
8.                 count += 1
9.         return count
```

```
10.     def isPrime(self,n:int)-> bool:  
11.         for i in range(2,n):  
12.             if n%i == 0: return False  
13.         else: return True  
14. #Driver code  
15. sol = Solution()  
16. print(sol.countPrimes(20))
```

**Output:** 8

**Check:** 2,3,5,7,11,13,17,19

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The outer loop runs  $n$  times, and in each iteration, `isPrime` function is called which also runs the loop  $n$  times. Thus, the time complexity is  $O(n^2)$ .

### **Space complexity:**

We don't require any extra space. So, the space complexity is  $O(1)$ .

## Approach 2 – Eratosthenes sieve method

We will use a clever algorithm called Eratosthenes sieve that avoids repetitive work. Let us create an array of integers from 0 to  $n$  named `prime`. The  $n^{\text{th}}$  element is `true` if  $n$  is prime. It will initially be marked as all primes. Let us illustrate this with an example where  $n=10$ . As 0 and 1 are not prime numbers, mark them as `False`.

Initial: Prime = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.  
F, F, T, T, T, T, T, T, T, T, T

Now, we will start with the smallest prime number, that is, 2, and mark all the multiples of 2 in the array as non-prime. Do not mark 2.

Step 1: Prime = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.  
F, F, T, T, F, T, F, T, F, T, F

Then we will take the next unmarked number, 3, and mark its multiples in the array as non-prime. 6 has been already marked, so mark 9.

Step 2: Prime = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

F, F, T, T, F, T, F, T, F, F, F

Then we will take the next unmarked number, 5, and mark its multiples in the array as non-prime. The only multiple in the range is 10, and it has already been marked. Then next unmarked number is 7 and it does not have any multiples in the range.

So, the list of primes up to 10 is 2, 3, 5, 7.

From the above example, we can note that for an unmarked number  $p$ , we need to start marking  $p^2$  onwards. The smaller ones would have been already marked. It also means that we can terminate the algorithm when  $p^2$  exceeds  $n$ . In our example, we could have terminated at  $p = 3$ . In the end, only the primes will remain True. We can simply count them.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def countPrimes(self, n: int) -> int:
3.         if n <= 1: return 0      #Base cases
4.         prime = [True for i in range(n)] #Create an array of
n True elements
5.         prime[0]=prime[1]=False  # We know 0 and 1 are not
primes
6.         p = 2
7.         while (p * p < n):    #Same as p < sqrt(n)
8.             # Check if p is prime so far
9.             if (prime[p] == True):
10.                 # Mark all multiples of p as non prime
11.                 for i in range(p * p, n, p):
12.                     prime[i] = False
13.                 p += 1
14.         return sum(prime)
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

It is difficult to calculate the time complexity. With the complicated calculations, it can be shown to be  $O(\log(\log(n)))$ .

### **Space complexity:**

We are creating an array of length  $n$ . So, the space complexity is  $O(n)$ .

## **3.7 Question 29 – How will you find the longest substring without the repeating characters?**

This question illustrates the use of a dictionary for string processing.

### **Problem statement**

You are given a string. Find the length of its longest substring which does not have repeating characters. Note that the substring means a slice of the original string without the permutation of the position.

Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:** “bcdcbc”

**Output:** 3

**Explanation:** The longest substring is “bcd”, which has the length 3.

#### **Example 2:**

**Input:** “aaaaa”

**Output:** 1

**Explanation:** The longest substring is “a”, with the length 1.

#### **Example 3:**

**Input:** “pwwkew”

**Output:** 3

**Explanation:** The longest substring is “wke”, with the length 3.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def lengthOfLongestSubstring(self, s: str) -> int:
```

## Strategy

We will study this with two approaches, the brute force, and the sliding window.

### Approach 1 – Brute force

First, let us enumerate all the substrings of the input string  $s$ . Then, we find the maximum length substring that does not contain duplicates. To find the existence of the duplicates, we will insert the substring characters in the set, named  $ss$ .

We will create two pointers,  $l$ (left) and  $r$ (right), that will enclose the substring. We will let  $l$  to run from 0 to the end of the input string in the outer loop. In the outer loop, we will first create an empty set  $ss$ . Then, we will start the inner loop and let  $r$  to run from  $l+1$  to the end of the input string. If the character pointed by  $r$ , that is,  $s[i]$  is not in the set  $ss$ , we will add it, update the  $longcount$ , and continue. If the new character exists in  $ss$ , we have found repetition and will exit the inner loop. In the end, the  $longcount$  contains the maximum length.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def lengthOfLongestSubstring(self, s: str) -> int:  
3.         if len(s) <= 1: return len(s) #Take care of s= "" and  
s="a"  
4.         longcount = 0 Initialize longest count to 0  
5.         for l in range(len(s)):
```

```

6.             ss = set()    #Create empty set to store
characters of substring
7.             for r in range(l, len(s)): # iterate right pointer
from left to end
8.                 if s[r] not in ss:
9.                     ss.add(s[r])
10.                    longcount = max(longcount, len(ss))
11.                else
12.                    longcount = max(longcount, len(ss))
13. #When you find duplicate, exit the inner loop
14.                 break
15.             return longcount
16. #Driver code
17. sol = Solution()
18. s = "abca"
19. print(sol.lengthOfLongestSubstring(s))

```

**Output:** 3

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Let  $n=\text{len}(s)$ . The outer loop runs  $n$  times. The inner loop runs  $n-i$  times. The total number of *runs* =  $n + (n-1) + (n-2) \dots + 1 = n(n+1)/2 = .5n^2+n$ . So, the time complexity is  $O(n^2)$ .

### **Space complexity:**

We create a set with the worst case length of  $n$ . So, the space complexity is  $O(n)$ .

## Approach 2 – Sliding window

Just like earlier, let us create two pointers to the window, namely `left` and `right`. Initially, both will be zero, indicating a closed window. The

difference is that the left pointer need not be increased by only 1, it can jump many steps at a time.

We will create a variable, named `ans` to store the maximum length. It will be initialized to 0. Then, we will create a dictionary, named `m` to remember where the latest occurrence of every character is. It will consist of the `element:index` pairs. We will open the window by shifting the right index to the right. We will load the element `s[right]` in a variable, named `el`. If `el` is not present in the dictionary `m`, we will add the pair `el:right` to `m`. If `el` exists in `m`, we have found a repetition. This is the end of a sub-sequence. Its length is given by `right-left+1`. If the length is more than `ans`, we will update the `ans`. The dictionary always keeps the latest position of `el`. So, we will move the left pointer next to the previous occurrence of `el`, that is, to `m[el]+1`, and replace `m[el]` with `right`.

Let us walk through the algorithm with an example. Suppose `s = "abcdefg"`

The initial conditions are shown in [Figure 3.1](#):

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
	↑↑								

left right

$m=\{\}$ ,  $left = 0$ ,  $right = 0$ ,  $answer = 0$ .

*Figure 3.1: Initial conditions*

In the first three iterations of our example, the new element is not in `m`. At the end of these three iterations, the variables are as shown in [Figure 3.2](#):

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
	↑	↑							

left      right

$m=\{a:0, b:1, c:2\}$ , answer =  $2-0+1 = 3$ , left = 0, right = 2.

*Figure 3.2: State after three iterations*

The fourth iteration starts with  $\text{right} = 3$ , so  $\text{el} = s[3] = a$ . This is present in  $m$  at index 0. We have broken the run which consisted of the sequence "abc". The length of the present run is reflected in the answer. It's time to start a new run from index 1. So we update left to 1. The new index of a is 3. So, the entry in  $m$  will get updated from  $a:0$  to  $a:3$ . Now, the current sequence is "bca" with the length 3. The state of the variables is shown in [Figure 3.3](#):

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
	↑	↑							

left      right

$m=\{a:3, b:1, c:2\}$ , answer = 3, left = 1, right = 3.

*Figure 3.3: State after four iterations*

The next three iterations add d, e, and f to the substring. The variable status at the end of the 7<sup>th</sup> iteration is shown in [Figure 3.4](#):

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
		↑				↑			
		left				right			

$m=\{a:3, b:1, c:2, d:4, e:5, f:6\}$ , answer =  $6-1+1=6$  (representing "bcadef")  
 left = 1, right = 6.

*Figure 3.4: State after 7 iterations*

In the 8th iteration, right=7. So,  $el = s[7] = c$ . Since c is present in m, we break the run again. This run was "bcadef". Now we know the procedure. The old position of c was 2. So, we start a new run from the next, that is, index 3. Entry c:2 in the dictionary m is updated to c:7.

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
			↑			↑			
			left			right			

$m=\{a:3, b:1, c:7, d:4, e:5, f:6\}$ , answer =  $\max(6,7-3+1)=6$  (representing "adefc"),  
 left = 3, right = 7.

*Figure 3.5: State after 8 iterations*

The status of the variables at the end of the 8<sup>th</sup> iteration is shown in [Figure 3.5](#).

In the 9<sup>th</sup> and the last iteration, right=8 and g are added.

The status of the variables at the end of the 9th iteration is shown in [Figure 3.6](#).

Index	0	1	2	3	4	5	6	7	8
S[index]	a	b	c	a	d	e	f	c	g
			↑					↑	
				left					right

**m={a:3, b:1, c:7,d:4, e:5, f:6, g:8}**

**answer** = max(6,8-3+1)=6 (representing “adefcg”)

**left = 3, right = 8.**

**Figure 3.6:** State after 9 iterations

The program returns answer = 6.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def lengthOfLongestSubstring(self, s: str) -> int:
3.         m = {} #Create empty dictionary
4.         left = 0 #Initialize left index to 0
5.         right = 0 #Initialize right index to 0
6.         ans = 0 #Initialize answer to 0
7.         n = len(s)
8.         for right in range(len(s)):
9.             el = s[right] #Get element at right index
10.            if(el in m):#If el is found in m
11.                left = max(left,m[el]+1) #Then slide left
index by 1
12.                m[el] = right #Update m
13.                ans = max(ans,right-left+1) #Update ans
14.        return ans
15.
16. sol = Solution()
```

```
17. s = "abcdefgc"  
18. print(sol.lengthOfLongestSubstring(s))
```

**Output:** 6

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Let the length of  $s$  be  $n$ . The *for* loop runs  $n$  times. The time complexity of search in the dictionary is  $O(1)$ . Hence, the overall time complexity is  $O(n)$ .

### **Space complexity:**

Since we are building a dictionary that can house  $n$  elements in the worst case, the space complexity is  $O(n)$ .

## 3.8 Question 30 – How will you convert a roman numeral into a decimal numeral?

This is an excellent example of text processing and utilizing the dictionary data structure.

### Problem statement

Write a program to convert a string representing a roman numeral into a decimal numeral. The roman numerals are represented by seven alphabets: I, V, X, L, C, D, and M. Their values are tabulated in [Table 3.1](#):

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

**Table 3.1:** Single letter roman symbols

The roman system is not positional. The total value of the number is the sum of the symbols. A symbol can be repeated up to three times. The symbols are normally written in decreasing order.

For example,  $MDC = 1000 + 500 + 100 = 1600$

But, if a smaller symbol is written before a larger symbol, then the value of the smaller symbol is negative.

For example,  $IV = -1 + 5 = 4$

Let us see some more examples.

**Example 1:** “III” = 3

**Example 2:** “IV” = 4

**Example 3:** “IX” = 9

**Example 4:** “LVIII” = 58

**Example 5:** “MCMXCIV” = 1994

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def romanToInt(self, s: str) -> int:
```

## Strategy

If the symbols appear only in decreasing order, it is a simple matter to add up their values. But a smaller symbol appearing before a bigger symbol becomes negative. This necessitates scanning the string back and forth. To avoid this, let us create some more symbols consisting of the two alphabets as shown in [Table 3.2](#):

Symbol	Value
IV	4
IX	9
XL	40
XC	90
CD	400

CM	900
----	-----

**Table 3.2:** Two-letter Roman symbols

A convenient way of converting the symbols to values is with the dictionary. The dictionary search happens with the time complexity of  $O(1)$ . We will create a dictionary roman for the single letter symbols, and roman2 for the double letter symbols. First, we will match the two-lettered symbols to two letters from the input strings.

The expression `s[i:i+2]` extracts a two element slice `s[i]` and `s[i+1]`.

The expression `roman2[s[i:i+2]]` fetches the numerical value of the two-letter symbol from [Table 3.2](#).

The expression `ans+=roman2[s[i:i+2]]` adds the value to the answer.

If none of the double letter symbols are matching, then we will try the single letter table roman. We will continue like this till the end of the string.

## Python code

The following code implements this strategy:

```

1. class Solution:
2.     def romanToInt(self, s: str) -> int:
3.         #Create dictionary to convert roman symbols to value
4.         roman =
5.             {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
6.         roman2 =
7.             {'IV':4, 'IX':9, 'XL':40, 'XC':90, 'CD':400, 'CM':900}
8.         i = 0
9.         ans = 0 #initialize answer to 0
10.        while i < len(s): #Scan the string with i
11.            if i<len(s)-1 and s[i:i+2] in roman2:#First try
12.                double letter symbols
13.                    ans+=roman2[s[i:i+2]]      #Add value to answer
14.                    i+=2
15.            else                                #Now try single letter
16.                symbols

```

```
13.         ans+=roman[s[i]]           #Add value to answer
14.         i+=1
15.     return ans
16. sol = Solution()
17. print(sol.romanToInt("III"))
18. print(sol.romanToInt("CDXLIII"))
```

### **Output:**

3  
443

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

The loop runs only once. Hence, the time complexity is  $O(n)$ .

### **Space complexity:**

We don't need any extra space. So, the space complexity is  $O(1)$ .

## **3.9 Question 31 – How will you identify a single number in an array?**

We will study various approaches to answer this math-related question.

### **Problem statement**

You are given an array of numbers, named `nums` in which every number appears twice except one, which appears only once. Identify this number. Let us illustrate this with the help of an example.

### **Example:**

**Input:** [2,5,1,2,5]

**Output:** 1

**Explanation:** The numbers 2 and 5 occur 2 times. The number 1 occurs only once.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def singleNumber(self, nums: List[int]) -> int:
```

## Strategy

We will study three approaches to solve this problem, using a list, using a hash table, and using an algebraic formula.

### Approach 1 – Using a list

We will create an empty list, named `uniques`. Then, we will scan the elements of the `nums` array. If the element is not present in the list, we will add it. If it is present, we will remove it. In the end, the list will contain only the element that occurred once.

### Python code

The following code implements this strategy:

```
1. from typing import List  
2. #List approach  
3. class Solution:  
4.     def singleNumber(self, nums: List[int]) -> int:  
5.         uniques = []  
6.         for i in nums:  
7.             if i not in uniques:  
8.                 uniques.append(i)  
9.             else:  
10.                 uniques.remove(i)  
11.         return uniques.pop()  
12. #Driver code  
13. sol = Solution()  
14. print(sol.singleNumber([2,5,1,2,5]))
```

**Output:** 1

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The main loop runs  $n$  times. In each iteration, we search the list with the time complexity  $O(n)$ . Hence, the overall time complexity is  $O(n^2)$ .

### **Space complexity:**

We are creating an array of length  $n$ . So, the space complexity is  $O(n)$ .

## Approach 2 – Using a hash table

To expedite the search, we will use a hash table, named counts to maintain the occurrence count of each number. We will use the defaultdict class from the collections module for creating the hash table because it does not raise exceptions when the key is absent. In each iteration, we execute the statement:

```
counts[i] += 1
```

If the number  $i$  is not present, the default value 0 is assumed. If it is present, the count is incremented. In the end, we will search the table to find the element that has the occurrence count of 1. This strategy is similar to what we used for question 25 to find the majority element.

## Python code

The following code implements this strategy:

```
1. from collections import defaultdict
2. class Solution:
3.     def singleNumber(self, nums: List[int]) -> int:
4.         counts = defaultdict(int)
5.         for i in nums:
6.             counts[i] += 1
7.
8.         for i in counts:
9.             if counts[i] == 1:
```

```
10.           return i
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The main loop runs  $n$  times. Updating the hash table has the time complexity of  $O(1)$ . Hence, the overall time complexity is  $O(n)$ .

### **Space complexity:**

We are creating an array of length  $n$ . So, the space complexity is  $O(n)$ .

## Approach 3 –Algebraic method

Consider the array  $\text{nums} = [\text{a}, \text{c}, \text{b}, \text{c}, \text{d}, \text{a}, \text{b}]$

Let us get a unique representation of  $\text{nums}$  named  $\text{nums\_unique}$  as  $[\text{a}, \text{b}, \text{c}, \text{d}]$

The double of the sum of the elements of  $\text{nums\_unique}$  is  $2\text{a}+2\text{b}+2\text{c}+2\text{d}$ .

The sum of the elements of  $\text{nums}$  is  $2\text{a}+2\text{b}+2\text{c}+\text{d}$ .

Thus, we get  $\text{d}$  by subtraction.

$$\text{d} = (2\text{a}+2\text{b}+2\text{c}+2\text{d}) - (2\text{a}+2\text{b}+2\text{c}+\text{d})$$

Python has two useful functions that will simplify the job for us. The first is `set`, which obtains a unique representation, and the second is `sum`, which obtains the sum of the elements of a set. With these functions, the Python function reduces to a single line.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def singleNumber(self, nums: List[int]) -> int:
3.         return 2 * sum(set(nums)) - sum(nums)
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The set created from the list has the time complexity of  $O(n)$ . The sum function has the time complexity of  $O(n)$ . Hence, the overall time complexity is  $O(n + n) = O(n)$ .

### **Space complexity:**

We are creating a set of length  $n/2$ . So, the space complexity is  $O(n/2)$ .

## **Conclusion**

We started the chapter with an introduction to the hash table and its implementation in Python using the dictionary and defaultdict. We also covered some general math-related formulae. We solved 9 questions to give you a solid foundation in this area. In the next chapter, we will cover the basic graphs, and in the following two chapters we will cover the traversal techniques, namely the DFS and the BFS.

## **Points to Remember**

- The hash table is an array-like object where the insertion and the search can be done with the  $O(1)$  time complexity. It is created by enclosing the items in curly brackets.
- The set is a dictionary with only the keys. It is also created by enclosing the items in curly brackets. But the empty set has to be created by using the `set()` function.
- The `Defaultdict` is a class derived from the dictionary. It does not throw exceptions when a non-existent key is tried.

## **MCQs**

1. What will be created by the instruction `a={}`?

- A. Empty dictionary
- B. Empty set
- C. Empty list

- D. Empty tuple
2. What is the output if we execute the set('Python') in the Python shell?
- A. {'P', 'y', 't', 'h', 'o', 'n'}
  - B. Python
  - C. {Python}
  - D. {'P', 'h', 'n', 'o', 't', 'y'}
3. What is the output if we execute the following commands in the Python prompt?
- ```
d={'maya':20,'ria':21}  
d
```
- A. 'maya' 20 ria 21
  - B. maya ria
  - C. {'maya':20,'ria':21}
  - D. 20 21
4. What is the output if we execute the following commands in the Python prompt?
- ```
d={'maya':20,'ria':21}  
d.get('piya',15)
```
- A. 20
  - B. 15
  - C. 'piya':15
  - D. Error
5. Suppose d={'maya':20,'ria':21}
- To delete the entry for 'maya', what command do we use?
- A. d.delete('maya')
  - B. d.delete('maya':20)
  - C. del d['maya':20]
  - D. del d['maya']

## Answers to the MCQs

**Q1:** A

**Q2:** D

**Q3:** C

**Q4:** B

**Q5:** D

## Questions

1. What is the time complexity of the dictionary search operation?
2. Explain dictionary, set, and defaultdict?
3. What are the two arguments in the “get” method of the dictionary?
4. How does a dictionary achieve the time complexity of  $O(1)$ ?
5. Explain the “sort” method of the list class.
6. Write the command to print the key corresponding to the maximum value in a dictionary.

## Key terms

Set, dictionary, get, defaultdict, sort.

## CHAPTER 4

### Trees and Graphs

The graph is an important data structure in Computer Science and it is a frequently asked topic in interviews. The tree is a special case of a graph. The Graph Theory has many practical applications in the fields of Electronics, Computer Science, Natural Sciences, and Social Sciences.

Most of the problems in the graph theory are based on traversing a graph. There are two methods for traversing a graph, namely the Depth First Search (DFS) and the Breadth First Search (BFS). The BFS algorithm traverses a graph in a breadth-wise manner. So it enters all the neighbors of a vertex in a queue. Initially, the starting vertex is entered in the queue. For trees, the root is the starting point. In the case of a graph, we need to select some arbitrary vertex as the starting node. When a dead-end occurs, we get the next vertex from the queue. The DFS algorithm is similar but it stores the neighbors in a stack instead of a queue. This results in a depth-wise search.

For traversing trees, the DFS has three variations, namely pre-order, post-order, and in-order. The BFS for tree traversal is called level order traversal.

### Structure

We will first revise the basic definitions of trees and graphs, and their representation in memory. Before moving on to questions, we will do some coding practice in Python for building and printing graphs and trees. These operations are typically required for writing the driver codes. Then, we will solve four general problems that cannot be classified as either DFS or BFS. The problems based on the DFS or BFS techniques will be discussed in the subsequent chapters.

We will cover the topics in the following order:

- Basics of graph theory

- Representation and manipulation of graphs
- Representation and manipulation of trees
- Recursion
- Question 32: How will you detect a redundant road connection?
- Question 33: How will you find the lowest common ancestor in a binary tree?
- Question 34: Who is the town judge?
- Question 35: How will you select flowers so that the adjacent gardens are not the same?

## **Objectives**

After studying this unit, you will have a solid base in graph theory. You will be able to write the graph manipulation code in Python, formulate word problems in terms of graphs, and get prepared for solving advanced problems involving the BFS and the DFS.

## **4.1 Basics of Graph Theory**

Let us review some basic definitions of graphs:

- A graph is a diagram consisting of points and lines that connect two points.
- A non-trivial graph must have at least one point. If it has only points and does not have any line, it is called a null graph. The points are called vertexes or nodes and the lines are called edges.
- The edges can be unidirectional (directed), or bidirectional (undirected). Optionally, the edges may have a parameter called weight associated with them. It represents the cost of traversing that edge.
- A simple graph allows only one edge between a pair of nodes and a multi-graph allows more than one edge between the same pair of nodes. A pseudo-graph allows an edge to start and end on the same node like a loop.
- If the edges of a graph are undirected, the graph is called undirected.

- The number of the edges that meet at a vertex of a undirected graph is called the degree of the vertex. A vertex with the degree 0 is called an isolated vertex, and a vertex with degree 1 is called a pendant vertex. If the edges are directed, we have to consider the in-degree and the out-degree separately.
- Graph  $G$  is denoted as  $G=(V,E)$ , where  $V$  is the set of vertices, and  $E$  is the set of edges.
- The number of edges in a undirected graph is equal to half the sum of degrees of its vertexes, i.e.,  $E = \frac{1}{2} \sum \deg(n)$
- The cycle is a path of connected edges that start and end on the same vertex.
- A graph is called cyclic if it has at least one cycle.
- A graph is called acyclic if it does not contain any cycle.
- A graph is called connected if there exists a path between every pair of the vertices.
- A graph is called complete or fully connected if there exists an edge between every pair of vertices. The number of edges is  ${}^nC_2 = n(n-1)/2$ .
- A graph is called disconnected if it has at least one pair of not connected vertices.
- A tree is a connected acyclic graph.
- The vertices and edges in a connected acyclic graph are related by the equation:  $V = E + 1$ .
- A binary tree is a tree in which a node cannot have more than two children, that is, it can have 0, 1, or 2 children.
- A tree node without parents is called a root node.
- A tree node with no children is called a leaf node.
- A forest is a set of unconnected trees.
- A subgraph of  $G$  is called a spanning tree if it has no cycles and has all the vertices of  $G$ .

## 4.2 Representation and Manipulation of Graphs

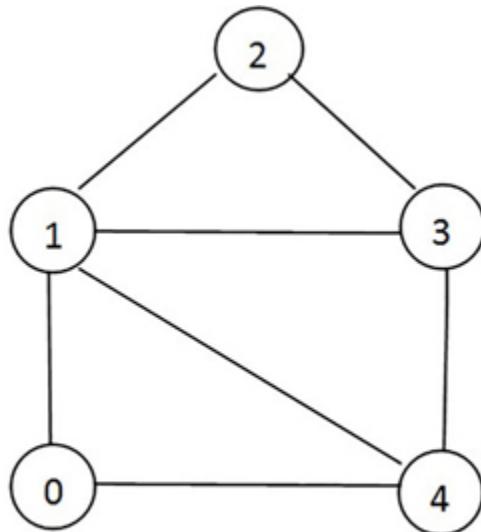
A graph is a nonlinear data structure. We will study how to represent it in memory. Then we will create a graph class that abstracts the functions of graph creation, adding edges, and printing.

### 4.2.1 Representation

There are two ways of representing a graph in memory:

1. Adjacency matrix
2. Adjacency list

Let the graph have  $V$  vertices labeled  $v_0, v_1 \dots v_{N-1}$ . The adjacency matrix  $A$  is the  $N \times N$  matrix, where element  $A[i][j] = 1$  if there is a path from  $v_i$  to  $v_j$ . The adjacency matrix of a undirected graph is symmetric, implying that if there is a path from  $v_i$  to  $v_j$ , there is also a path from  $v_j$  to  $v_i$ . Consider the graph in [Figure 4.1 \(a\)](#). Its adjacency matrix representation is shown in [Figure 4.1 \(b\)](#):



(a) Example graph

$\downarrow i \rightarrow$	$j \rightarrow$	0	1	2	3	4
0	0	1	0	0	1	
1	1	0	1	1	1	
2	0	1	0	1	0	
3	0	1	1	0	1	
4	1	1	0	1	0	

(b) Adjacency matrix of (a)

*Figure 4.1: Adjacency matrix representation of a graph*

The adjacency matrix representation is very convenient for data manipulation but very wasteful on memory. Most practical graphs are sparse, that is, they have a very small number of edges.

The adjacency list is an efficient method for representing sparse graphs. Let us consider the adjacency matrix to be an array of rows. Each row corresponds to a vertex. Instead of listing every other vertex as 0 or 1, we will store a list of connected vertices only. It need not be a linked list. It could be a simple dynamic array which is called a list in Python. But sometimes, the vertex identifiers may not be a continuous sequence of integers. The vertex id may be any arbitrary string. In this case, it is better to implement it with a dictionary where the vertex ID is the key and the destination list is the value. This way, searching for a row becomes fast. The normal dictionary that we create with curly brackets {} has one undesirable property. It throws an exception when we try to search a non-existent key. If we don't handle the exception, the program gets terminated. To avoid this problem, we will use a defaultdict class defined in the collections library of Python. This dictionary class returns a default value when a key is not found, rather than throwing an exception.

The adjacency list of the graph of [Figure 4.1](#) is:

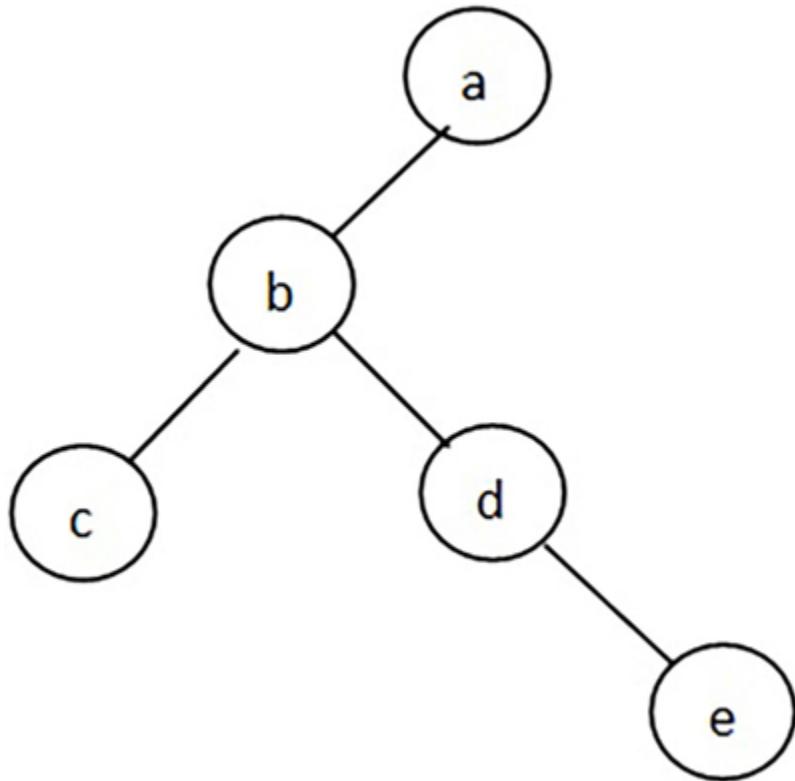
- 0: [1,4]
- 1: [0,2,3,4]
- 2: [1,3]
- 3: [1,4]
- 4: [0,1,3]

### 4.2.2 Manipulation of graphs

We will create a class, named Graph to abstract the functionality of a undirected graph. It will be represented in the adjacency list form. We will implement the functions for initializing, adding an edge, and printing the graph.

In the initialization function of the graph, we will instantiate an object of defaultdict(list) class named graph. This will be the storehouse of the graph information. Function insertEdge(self,v1,v2) adds vertex v2 to row v1 and adds vertex v1 to row v2. The function printGraph prints all rows.

The driver code creates the graph shown in [Figure 4.2](#):



*Figure 4.2: Example graph*

## Python code

The following code implements this strategy:

```
1. from collections import defaultdict
2. class Graph:
3.     def __init__(self):
4.         self.graph = defaultdict(list)#Create empty
dictionary
5.
6.     def insertEdge(self, v1, v2):
7.         self.graph[v1].append(v2)#Add v2 to list of v1
8.         self.graph[v2].append(v1)#Add v1 to list of v2
9.
10.    def printGraph(self):
11.        for node in self.graph:
12.            print(node,':',end=' ')#print vertex-id:
```

```

13.             for v in self.graph[node]:#print every vertex
in the list
14.                 print(v,end=' ')
15.                 print('\n')#print new line at end of the list
16.
17. #Driver code
18. g = Graph()
19.
20. g.insertEdge('a', 'b')
21. g.insertEdge('b', 'c')
22. g.insertEdge('b', 'd')
23. g.insertEdge('d', 'e')
24.
25. g.printGraph()

```

### **Output:**

```

a : b
b : a c d
c : b
d : b e
e : d

```

## **4.3 Representation and Manipulation of Trees**

Recall that a tree is an acyclic graph. The nodes in a binary tree can have a maximum of two children. The binary trees are widely used in algorithms. Before going on to the questions, we will first study the representation of binary trees in memory and then write two helper functions. The first is for creating a binary tree from an array, and the second is for printing a binary tree.

### **4.3.1 Representation of binary trees in memory**

We will use the following class to capture the functionality of a binary tree node:

```

1. class TreeNode:
2.     def __init__(self, v):
3.         self.val = v
4.         self.left = None
5.         self.right = None

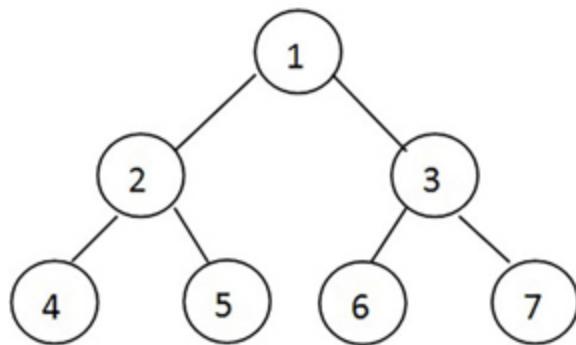
```

In most of our examples, we consider `val` to be a single entity like an integer but it could be a complex data structure as well. Each node stores references to the children, namely left and right.

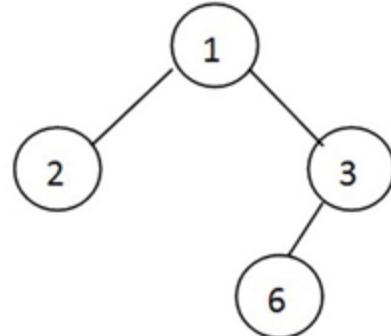
A tree will be represented by a collection of nodes with proper connections in the links.

For manual entry, we often use the array representation. Starting with the root node, the data of the nodes are entered level-by-level in a linear array. There is no marker indicating a change of the level. Therefore, the number of items in the  $k^{\text{th}}$  level has to be always  $2k$ . Consider the complete tree is shown in [Figure 4.3 \(a\)](#). The corresponding array is [1,2,3,4,5,6,7]. The first level has only 1 element, [1]. The second level has two elements, [2, 3]. The third level has four elements, [4, 5, 6, 7].

If the tree is not complete, we need to enter null elements as place holders to meet the number  $2k$ . For the incomplete tree in [Figure 4.3 \(b\)](#), the array is [1,2,3,None,None,6,None]:



(a)Complete tree



(b)Incomplete tree

*Figure 4.3: Example of trees*

If the index of a parent node data in the array is  $p$ , the index of the left child data is  $2p$ , and the index of the right child data is  $2p+1$ . If the index of the

child node data in the array is  $c$ , the index of the parent data is  $c//2$ .

### **4.3.2 Creation and display**

Many times, we need to create a tree as input data in the driver code. One simple way is to instantiate nodes as objects of the `TreeNode` class and to establish relations manually.

#### **Python code**

The manual method code is as follows:

```
1. root = TreeNode(1)
2. root.left = TreeNode(2)
3. root.right = TreeNode(3)
4. root.left.left = TreeNode(4)
5. root.left.right = TreeNode(5)
6. root.right.left = TreeNode(6)
7. root.right.right = TreeNode(7)
```

This code creates a tree as shown in [Figure 4.3 \(a\)](#). But we will now define a function `makeTree` to make the job easier. The template of `makeTree` will be:

```
def makeTree(arr:List[int])-> TreeNode:
```

The problem of creating the tree is that the array is scanned from left to right. When we want to fill the left and right fields of an  $i^{\text{th}}$  node, the child nodes  $2i+1$  and  $2i+2$  are yet to be created. One way of overcoming the problem is to create an array of  $n$  nodes and fill only the `val` field. Then run another loop to fill the left and right pointers. But this is wasteful on memory. We don't want to create another data structure for storing the addresses of tree nodes.

The problem can be solved if we use recursion.

We will define a recursive function `NodeInsert` which takes in a null node as input and returns the node after filling its left and right pointers. Other

arguments of the `NodeInsert` are the array `arr` and `i`, the pointer in the `arr`. The template of `NodeInsert` is as follows:

```
def NodeInsert(arr:List[int],root:TreeNode,i:int)-> TreeNode:
```

The function instantiates a node of `TreeNode` class with `val = arr[i]` and assigns it to the root. Its left and right pointers are `None` at this time. The function recursively calls itself to fill these pointers. Exit condition from the recursion is met when `i` equals or exceeds `n`.

Suppose the `makeTree` function is invoked with the input `[1, 2, 3, 4, 5, 6, 7]`. It calls `NodeInsert` with parameters `arr = [1, 2, 3, 4, 5, 6, 7]`, `root = None`, and `i=0`.

The function creates a node with `val = arr[0] = 1`, and calls itself with `i = 1`.

As the recursion progresses, `i` goes through values 0-1-3-4-2-5-6. Finally, it returns the root of the tree.

## Python code

The following code implements this function:

```
1. def makeTree(arr:List[int])-> TreeNode:  
2.     return NodeInsert(arr,None,0)  
def NodeInsert(arr:List[int],root:TreeNode,i:int)-> TreeNode:  
3.     n=len(arr)  
4.     if i<n:  
5.         root = TreeNode(arr[i])  
6.         # insert left child  
7.         root.left = NodeInsert(arr, root.left, 2 * i + 1)  
8.         # insert right child  
9.         root.right = NodeInsert(arr, root.right, 2 * i + 2)  
10.    return root#Driver code  
11. root = makeTree([1,2,3,4,5,6,7])
```

Now, we need to find out if the tree has been properly created. For this purpose, we will write a function `printTree`. We want to print the nodes

level-wise. So we have to use the level order traversing strategy. It is also called the BFS. The `printTree` function uses a recursive helper function `printLevel`. The `printLevel` function takes in a node list of one level as input, prints it, and returns the node list of the next level. It keeps on calling itself as long as the output list contains at least one non-null element.

## Python code

The Python code is as follows:

```
1. def printTree(root:TreeNode)-> None:  
2.     LevelList = [root]  
3.     printLevel(LevelList)  
4.  
5. def printLevel(LevelList:List[TreeNode])-> List[TreeNode]:  
6.     LevelStr = ""           #Initialize this level string  
7.     outList = []            #Initialize the output list  
8.     ListEmpty = True  
9.     for node in LevelList:  
10.         if node is None:  
11.             LevelStr += "None"  
12.             outList.append(None)  
13.             outList.append(None)  
14.         else:  
15.             LevelStr += (str(node.val) + " ")  
16.             outList.append(node.left)  
17.             outList.append(node.right)  
18.             ListEmpty = False  
19.         if not ListEmpty:    #When we reach lowest level, list  
is empty  
20.             print(LevelStr)  
21.             printLevel(outList)  
22. #Driver code  
23. printTree(root)
```

## **Output:**

```
1  
2 3  
4 5 6 7
```

## 4.4 Recursion

In the graph theory, we find recursive functions very useful. We will study the general principles of recursion so that we can apply recursion for the solutions to the problems.

Recursion is a process in which a function calls itself. It may call itself directly or indirectly. If function *A* calls function *B* and function *B* calls function *A*, it is an indirect call. Many algorithms can be best described by a recursive function.

A well-defined recursive function must satisfy two conditions:

1. There must be a base case for which the function does not call itself. It provides an escape path from an infinite loop.
2. In every iteration, the argument moves closer to the base case by means of a recurrence relation which builds a new solution from the previous solutions.

This technique makes coding simple for some problems, like towers of Hanoi, DFS, tree traversal, backtracking, and so on.

For each call, the memory for local variables is created on the stack, so they are preserved on return from a recursive call. Let us take this example to understand this process clearly:

```
1. def printFunc(n):  
2.     if (n < 1):  
3.         return  
4.     else:  
5.         print(n, end = " ") # 1  
6.         printFunc(n-1)      # 2  
7.         print(n, end = " ") # 3  
8.     return  
9.
```

10. `printFunc(3)`

**Output:** 3 2 1 1 2 3

Let us understand the output.

- **First invocation:** When `printFunc(3)` is called from the main, a local variable  $n$  is created to store value 3 and it is printed by statement 1. Statement 2 calls `printFunc(2)`.
- **Second invocation:** Statement 2 of the first invocation calls `printFunc(2)`. In this call,  $n$  is 2. It is printed. Statement 2 calls `printFunc(1)`.
- **Third invocation:** Statement 2 of the second invocation calls `printFunc(1)`. In this call,  $n$  is 1. It is printed. Statement 2 calls `printFunc(0)`.
- **Fourth invocation:** Statement 2 of the second invocation calls `printFunc(0)`. In this call,  $n$  is 0. Since this is less than 1, the function returns to the third invocation.
- **Return to the third invocation:** The value of  $n$  is 1. It is printed by statement 3 and the function returns.
- **Return to the second invocation:** The value of  $n$  is 2. It is printed by statement 3 and the function returns.
- **Return to the first invocation:** The value of  $n$  is 3. It is printed by statement 3 and the function returns to the main program.

Can you guess what will be the output of `printFunc(5)`? It is:

5 4 3 2 1 1 2 3 4 5

## 4.5 Question 32 – How will you detect a redundant road connection?

Consider that there are  $N$  cities in a country that need to be connected by modern highways. Several city-to-city construction proposals have been submitted to the Ministry. Due to the shortage of funds, the Ministry decides that only the minimum highways will be sanctioned such that no

city will remain unconnected. However, one redundant proposal has been erroneously sanctioned, that is, the cities will still be connected even if we remove this proposal. Can we identify it?

This problem exposes you to the union of a disjoint set algorithm.

## **Problem statement**

Consider a graph having vertices numbered from 1 to  $N$ . The graph is described in the form of a list of edges. One of the edges is redundant to form a tree, that is, this edge forms a cycle in the graph. If you remove this edge, there will be no cycles, that is, the graph will become a tree. You need to identify this edge. The extra edge may not be unique. Find any edge that removes a cycle. If there is no redundant edge, then return None. Let us illustrate this with the help of two examples.

**Example 1:**

**Input:** `[[1, 2], [1, 3], [2, 3]]`

**Output:** `[2, 3]`

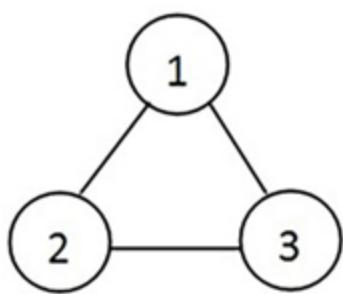
**Explanation:** In [\*Figure 4.4 \(a\)\*](#), edge `[2, 3]` is redundant.

**Example 2:**

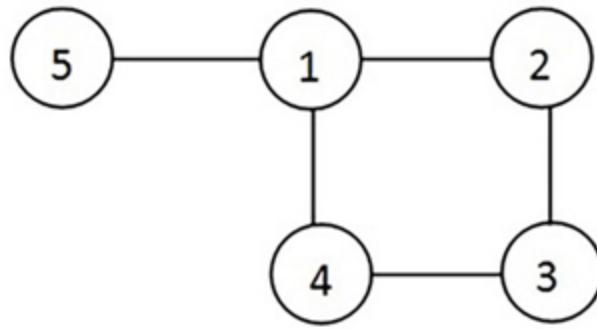
**Input:** `[[1, 2], [2, 3], [3, 4], [1, 4], [1, 5]]`

**Output:** `[1, 4]`

**Explanation:** In [\*Figure 4.4 \(b\)\*](#), edge `[1, 4]` is redundant.



(a)



(b)

**Figure 4.4:** (a) Example 1 (b) Example 2 graphs

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def findRedundantConnection(self, edges:  
List[List[int]]) -> List[int]:
```

## Strategy

The detection of a redundant edge is very easy. If  $E > (V-1)$ , there is a redundant edge. The identification needs some more work. The simplest way to solve this problem is to use the union of disjoint sets methodology. We will start with a set of  $N$  nodes with no edges at all. We will imagine that each node belongs to a set of connected nodes. The connected set will be identified by one root node, which is the parent of all nodes in the set. We will create an array named  $p$  having  $N$  elements to store the parent of each node. Initially, there are no edges, so all nodes are root nodes of their own connected sets. In other words, each node is the parent to itself. So the essential test to see whether or not a node is a root node is that  $p[i]$  should be equal to  $i$ . The initial contents of  $p$  will be  $[1,2,3,4\dots N]$ .

At this stage, we will add an *edge*  $(x,y)$  to the graph, so that the nodes  $x$  and  $y$  get connected. This operation is called a union. Arbitrarily, we will make one of them the parent of the other node. In our code, we will make  $y$  the parent of  $x$ . So now  $x$  and  $y$  belong to a set identified as  $y$ . As we go on adding edges, the vertices  $x$  and  $y$  of the new edge may either be roots or children. We need to find if they belong to the same set, that is, if they are descendants of the same root. For this purpose, we will write a function *find*, which recursively calls itself till we find that the parent of a node is the same as itself. The union function checks if the roots of  $x$  and  $y$  are the same. If they are the same, then we have found the redundant edge. We will return it as the answer. If the parents are different, then we will merge the two sets, that is, we will make the root of  $y$  as the parent of the root of  $x$ .

The outer loop simply scans the input list of edges and calls the union function. We need to estimate the number of nodes for creating the array  $p$ . A connected graph having  $E$  edges can have at the most  $E+1$  nodes. As the vertex numbering starts from 1, let us have  $E+2$  elements in the array  $p$ .

Let us walk through the example 2, where edges = [[1,2], [2,3], [3,4], [1,4], [1,5]]

Initially  $p = [0, 1, 2, 3, 4, 5, 6]$

1. In the first iteration of the *for* loop, the edge is [1,2]. It calls the union function which finds that 1 and 2 are roots. As these are different, it makes 2 as the parent of 1, that is,  $p[1]$  becomes 2. So,  $p$  becomes [0, 2, 2, 3, 4, 5, 6].
2. In the second iteration, the edge is [2,3]. It calls the union function which finds that 2 and 3 are the root nodes. Since roots are different, it makes 3 as the parent of 2, that is,  $p[2]$  becomes 3. Now,  $p$  becomes [0, 2, 3, 3, 4, 5, 6].
3. In the third iteration, the edge is [3,4]. It calls the union function which finds that 3 and 4 are the root nodes. Since the roots are different, it makes 4 as the parent of 3, that is,  $p[3]$  becomes 4. Now,  $p$  becomes [0, 2, 3, 4, 4, 5, 6].
4. In the fourth iteration, the edge is [1,4]. It calls the union function which finds that 2 is the parent of 1, 3 is the parent of 2, and 4 is the parent of 3. It also finds that 4 is a root node. Thus, the parents of 1 and 4 are the same. So, the union function returns *True*.

The *for* loop is terminated and the function returns [1,4]. The last edge [1,5] is not even read.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def findRedundantConnection(self, edges:
List[List[int]]) -> List[int]:
4.         #Create a parent array. Initially each node is its
own parent
5.         p = [x for x in range(len(edges)+2)]
6.
```

```

7.         #Function to return parent of x
8.         #Recursively call itself till we find that parent of
x is x itself
9.         def find(x):
10.             if p[x] != x: p[x] = find(p[x])
11.             return p[x]
12.         #If parents of x and y are different make y as
parent of x
13.         #Returns true if parents are same
14.         def union(x, y):
15.             if find(x) == find(y): return True
16.             p[find(x)] = find(y)
17.         #For source destination pairs of edge
18.         #if both have same parent return the pair x,y
19.         for x, y in edges:
20.             if union(x,y): return [x, y]
21. #Driver code
22. sol=Solution()
23. edges = [[1,2], [1,3], [2,3]]
24. print(sol.findRedundantConnection(edges))
25. edges = [[1,2], [1,3]]
26. print(sol.findRedundantConnection(edges))
27. edges =[[1,2], [2,3], [3,4], [1,4], [1,5]]
28. print(sol.findRedundantConnection(edges))

```

## **Output:**

[2, 3]

None

[1, 4]

## **Complexity**

The complexity analysis is as follows:

**Time complexity:** The number of vertices is  $E+1$ . The *for* loop runs  $E+1$  times. The two find calls can take maximum  $E+1$  iterations. Thus, the time complexity is  $O((E+1)*2*(E+1))$  which can be simplified as  $O(E^2)$ .

**Space complexity:** As we are creating the array  $p$ , the space complexity is  $O(E)$ .

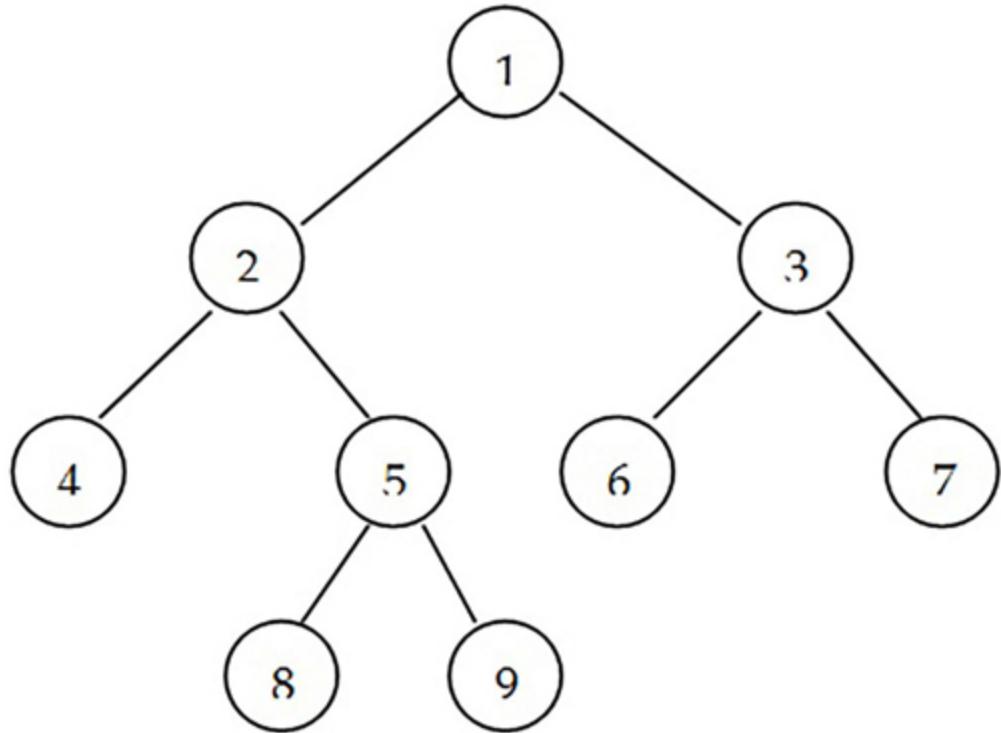
## 4.6 Question 33 – How will you find the lowest common ancestor in a binary tree?

Imagine a company with strict hierarchical rules, that is, an employee can receive instructions only from his immediate boss. So if an employee, Tom, wants something to be done by a colleague, Dick, in another department, he must tell his boss, he will tell his boss, and so on till we find a person, Harry, who is above Dick in the hierarchy. Then, he will send instructions down the line till they reach Dick. We don't want the issue to escalate to the CEO level each time, so we need to find the lowest common bosses of Tom and Dick. For simplicity, assume that one boss has only two subordinates. The situation can now be modeled by a binary tree.

We also allow a node to be a descendant of itself.

### Problem statement

Given the root of a tree and the data in any two nodes, find their lowest common ancestor and print its value. Consider the tree [1,2,3,4,5,6,7,null,null,8,9] depicted in [Figure 4.5](#). (Please refer to section 4.3.1 for the array representation of a tree):



**Figure 4.5:** Example tree

Let us illustrate this with the help of two examples related to [Figure 4.5](#):

**Example 1:** Input = root, n8, n6

**Output:** 1

**Explanation:** The lowest common ancestor of n8 and n6 is n1, with the value 1.

**Example 2:** Input = root, n4, n5

**Output:** 2

**Explanation:** The lowest common ancestor of n4 and n5 is n2, with the value 2.

## Solution format

Your solution should be in the following format.

```

1. class Solution:
2.     def lowestCommonAncestor(self, root: 'TreeNode', p:
   'TreeNode', q: 'TreeNode')      -> 'TreeNode':

```

## Strategy

If the tree node had a pointer to the parent, then the problem would have been very simple. We would have just followed these two steps:

1. Climb up the ancestry of  $p$  and store all the ancestors in some data structure (array, linked list, set, or dictionary).
2. Climb the ancestry of  $q$ , each time checking whether or not the ancestor exists in the list of ancestors of  $p$ . When we find a match, we have found the answer.

To make the search fast, it is better to choose the set as the data structure for storing ancestors of  $p$ .

Now let us supply the missing step, that is, the ancestry dictionary of all nodes. The dictionary named `parent` will consist of a node, parent pairs. Initially, it will be empty. Then starting from the root, we will traverse the tree and for each child node, we will enter the child: parent pair in the dictionary. As each iteration creates two nodes for future visiting (left and right children), we will insert the child nodes in a stack. We will pop one element from the stack and process it. Initially, the stack will consist of only the root. It is not necessary to traverse the whole tree. Once we have located  $p$  and  $q$ , our job is done. To keep track of it, we will create two flags, `p_not_found`, and `q_not_found`. As long as any one of them is true, we will keep looping.

Note that the inputs to the function, that is, `root`, `p`, and `q` are all pointers to the nodes, not node values. Therefore, the driver code that creates the test tree should give names to the pointers as shown below:

```
root.left = n2=TreeNode(2)
```

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':  
3.         # Create Stack of nodes to be traversed
```

```

4.         stack = [root]      #Initially it only has root
5.         # Create a Dictionary to store traversed parent
pointers
6.         parent = {root: None}
7.         p_not_found = True
8.         q_not_found = True
9.         # Keep looping until both p and q are added parent
dictionary
10.        while p_not_found or q_not_found:
11.            node = stack.pop()
12.            if node == p: p_not_found = False
13.            if node == q: q_not_found = False
14.            # If child exists, add it to stack and mark
node as its parent
15.            if node.left:
16.                parent[node.left] = node
17.                stack.append(node.left)
18.            if node.right:
19.                parent[node.right] = node
20.                stack.append(node.right)
21.        #Parent dictionary is ready
22.        # Create Ancestors set for node p.
23.        ancestors = set()
24.        # Find all ancestors of p using parent pointers.
25.        while p:
26.            ancestors.add(p)
27.            p = parent[p]
28.        # Ascend hierarchy of q. The first ancestor of q
which appears in
29.        # p's ancestor set must be their lowest common
ancestor.
30.        while q not in ancestors:
31.            q = parent[q]
32.        return q

```

```
33. root = n1=TreeNode(1)
34. root.left = n2=TreeNode(2)
35. root.right = n3=TreeNode(3)
36. root.left.left = n4= TreeNode(4)
37. root.left.right = n5=TreeNode(5)
38. root.right.left = n6=TreeNode(6)
39. root.right.right = n7=TreeNode(7)
40. n5.left=n8=TreeNode(8)
41. n5.right=n9=TreeNode(9)
42. printTree(root)
43.
44. sol = Solution()
45. print("ans ",sol.lowestCommonAncestor(root,n8,n6).val)
46. print("ans ",sol.lowestCommonAncestor(root,n4,n5).val)
47. print("ans ",sol.lowestCommonAncestor(root,n6,n7).val)
```

### **Output:**

```
1
2
3
```

### **Complexity Analysis**

The complexity analysis is as follows:

**Time complexity:** As we are visiting all the nodes once, the time complexity is  $O(n)$ .

**Space complexity:** As we are creating two dictionaries, the space complexity is  $O(n)$ .

### **4.7 Question 34 – Who is the town judge?**

This is a simple exercise of formulating a word problem in terms of graph theory. Once you model it properly, the solution is simple.

### **Problem statement**

There is a town, having  $N$  people. They are labeled from 1 to  $N$ . There is a rumor that one of these people is secretly the town judge. If the town judge really exists, then:

1. The town judge does not trust anybody.
2. Everybody (except for the town judge himself) trusts the town judge.
3. There is exactly one person that satisfies both the conditions 1 and 2.

Somebody conducted a survey in which people were asked to name one person that they trust. The result of the survey is available in the form of an array named `trust`. It is a list of truster:trustee pairs. In other words, `trust` is a list of lists, and:

If `trust[i] = [n1, n2]`

It means that person  $n1$  trusts person  $n2$ .

Write a function to identify the judge if one exists. If there is no judge, return -1. You must take care of the following base cases:

When  $N=1$  and `trust = []` judge is 1.

When  $N>1$  and `trust = []` judge does not exist.

Let us illustrate this with the help of two examples.

**Example 1:** Input:  $N = 4$ , `trust = [[1, 3], [1, 4], [2, 3], [2, 4], [4, 3]]`

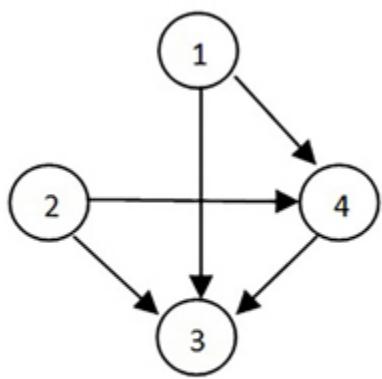
**Output:** 3

**Explanation:** The tree form of the trust array is shown in [Figure 4.6\(a\)](#). It is observed that there is no outgoing edge from node 3. So, 3 is the judge.

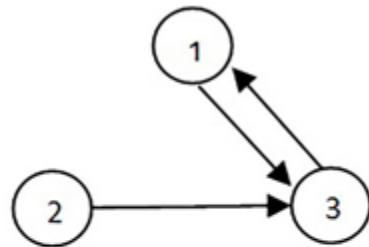
**Example 2:** Input:  $N = 3$ , `trust = [[1, 3], [2, 3], [3, 1]]`

**Output:** -1

**Explanation:** The tree form of the trust array is shown in [Figure 4.6 \(b\)](#). It is seen that every node has an outgoing edge. So there is no judge.



(a)



(b)

**Figure 4.6:** (a) Example 1 (b) Example 2

## Solution format

Your solution should be in the following format:

```

1. class Solution:
2.     def findJudge(self, N: int, trust: List[List[int]]) ->
int:

```

## Strategy

From the graphical illustration of [Figure 4.6](#), it is clear that the problem can be modeled as a directional graph. Each entry  $[n_1, n_2]$  in the trust represents an edge going from  $n_1$  to  $n_2$ . Further, we note that a valid judge has no outgoing edge and has  $N-1$  incoming edges. Thus we need to find the in-degree and out-degree of every node and then search for a node having an in-degree of  $N-1$  and an out-degree of 0.

First, we will handle the base cases of empty trust array. Then, we will create two arrays named `in_degree` and `out_degree` to count the incoming and outgoing edges. They will be initialized with  $N+1$  zeroes. One extra element is required because the numbering starts from 1. We will scan the trust array and for every edge, the `in_degree` of the source node and `out_degree` of the destination node is incremented.

Then we will scan the `in_degree` and `out_degree` arrays to find a node having the properties of a judge.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def findJudge(self, N: int, trust: List[List[int]]) ->
int:
4.         if len(trust)==0:                      #Base cases
5.             if (N == 1): return 1
6.             else: return -1
7.         in_degree = [0] * (N+1)
8.         out_degree = [0] * (N+1)
9.         for edge in trust:
10.             in_degree[edge[1]] += 1
11.             out_degree[edge[0]] += 1
12.             for i in range(len(in_degree)):
13.                 if (in_degree[i] == N-1) and (out_degree[i] ==
0):
14.                     return i
15.             return -1
16. #Driver code
17. sol = Solution()
18. print("Judge is ",sol.findJudge(4,[[1,3],[1,4],[2,3],[2,4],
[4,3]]))
19. print("Judge is ",sol.findJudge(2,[[1,2]]))
20. print("Judge is ",sol.findJudge(3,[[1,3],[2,3],[3,1]]))
21. print("Judge is ",sol.findJudge(2,[]))
```

## **Output:**

Judge is 3

Judge is 2

Judge is -1

Judge is -1

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** As we are running two loops of length  $n$ , the time complexity is  $O(n)$ .

**Space complexity:** As we are creating two arrays of length  $n$ , the space complexity is  $O(n)$ .

## 4.8 Question 35 – How will you select flowers so that the adjacent gardens are not the same?

This is another exercise of formulating a word problem in terms of graph theory. Actually, it is a variation of the map coloring problem. The use of a dictionary and set makes the solution simple.

### Problem statement

You have four types of flowers labeled 1,2,3,4. You want to plant them in  $N$  gardens, labeled 1 to  $N$ . The gardens are interconnected by many paths. The paths are described by a list of pairs  $[x,y]$ . An entry  $[x,y]$  in the list indicates that a bidirectional path exists between the garden number  $x$  and the garden number  $y$ . A garden has, at the most, three paths coming into or leaving it. The landscaping architect has advised that the flowers planted in connected gardens should be different.

Write a program to choose the flowers to be planted in each garden. The answer will be an array where  $\text{answer}[i]$  is the type of flower planted in the  $(i+1)^{\text{th}}$  garden. You may assume that a valid answer always exists. Any answer which will follow the architect's advice is valid. Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:**  $N = 4$ , paths =  $[[1,2],[2,3],[3,1],[3,4]]$

**Output:**  $[1,2,3,1]$

**Explanation:** Gardens 1, 2, and 3 are completely connected. So we need three different flowers, namely 1, 2, and 3. Gardens 1 and 4 are not connected. So, we can repeat flower 1.

### Example 2:

**Input:** N = 4, paths = [[1,2],[3,4]]

**Output:** [1,2,1,2]

**Explanation:** Gardens 1, 2 are connected to each other, and gardens 3, 4 are also connected to each other. There is no other connection. So we need only two different flowers, namely 1, 2.

### Example 3:

**Input:** N = 4, paths = [[1,2],[2,3],[3,4],[4,1],[1,3],[2,4]]

**Output:** [1,2,3,4]

**Explanation:** Gardens 1, 2, 3, 4 are completely connected. So we need four different flowers, namely 1, 2, 3 and 4.

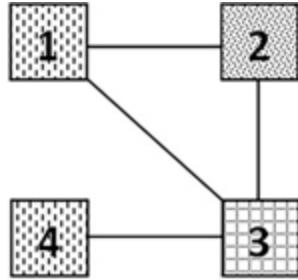
## Solution format

Your solution should be in the following format:

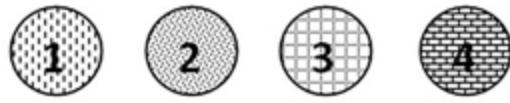
```
1. class Solution:  
2.     def gardenNoAdj(self, N: int, paths: List[List[int]]) ->  
List[int]:
```

## Strategy

There are  $N$  nodes and their interconnecting edges are given in the path list. The problem can be modeled as an undirected graph. The graph for the first example is shown in [Figure 4.7 \(a\)](#). The flower types are shown in [Figure 4.7 \(b\)](#). The gardens in [Figure 4.7 \(a\)](#) are filled with a pattern matching with the chosen color in the answer:



(a) Gardens and paths



(b) Colour legend

*Figure 4.7: Example 1 graph*

We will create an array of  $N$  integers, named `flowers` to store the allocation of flowers to the gardens, that is, this will be the answer. Initially, all the elements will be 0. As the garden list is 1 based but the list of the flowers is 0 based, `flowers[i-1]` contains the type of flower allocated to the garden number  $i$ .

Our next job is to create an adjacency list type of graph from the paths list. It will be a dictionary with the key as a node and the value as the neighbor list. It will be named `PathGraph`. Instead of using a simple dictionary, we will use `defaultdict` from the `collections` module because it does not throw an error if a non-existent key is presented.

The paths list is scanned in a loop. Each entry contains a `[node1, node2]` pair. `Node2` is added to the adjacency list of `node1` and `node1` is added to the adjacency list of `node2`. At this stage, the `PathGraph` looks like this:

```
1:[2,3]
2:[1,3]
3:[2,4,1]
4:[3]
```

Now comes the job of assigning flower types to gardens. We will run a loop with  $i$  spanning from 1 to  $N$ . (The range in Python has to be specified as  $1, N+1$ ). The variable  $i$  represents the garden number. We will create a set named `types`, to store the flower types available for planting. The data type set was chosen for a quick lookup. Initially, all the types are available, so the type contains 1,2,3,4.

Then we will scan the neighbors' list of the  $i^{\text{th}}$  garden. We will check whether or not the type allocated to the neighbor (that is, `flowers[neighbor -1]`) exists in types. If it does, it is not available. So, we will remove it from the `types`. Finally, types are left with a list of unallocated types. Now we will allocate the head of the unallocated list to the garden number  $i$ , that is, the pop the left element from `types` and store it in `flowers[i-1]`. (Recall that `flowers[i-1]` contain the type of flower in garden  $i$ .)

In the end, the flower array is returned.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. from collections import defaultdict
3. class Solution:
4.     def gardenNoAdj(self, N: int, paths: List[List[int]]) ->
List[int]:
5.         flowers = [0]*N
6.         pathgraph = defaultdict(list)
7.         for node1,node2 in paths:
8.             pathgraph[node1].append(node2)
9.             pathgraph[node2].append(node1)
10.            for i in range(1, N+1):
11.                types = set([1,2,3,4])
12.                for neighbor in pathgraph[i]:
13.                    if flowers[neighbor -1] in types:
14.                        types.remove(flowers[neighbor-1])
15.                flowers[i-1] = types.pop
16.            return flowers
17. #Driver code
18. sol=Solution()
19. print(sol.gardenNoAdj(4,[[1,2],[2,3],[3,4],[1,3]]))
20. print(sol.gardenNoAdj(3,[[1,2],[2,3],[3,1]]))
```

```
21. print(sol.gardenNoAdj(4, [[1,2],[3,4]]))  
22. print(sol.gardenNoAdj(4, [[1,2],[2,3],[3,4],[4,1],[1,3],  
[2,4]]))
```

### **Output:**

```
[1, 2, 3, 1]  
[1, 2, 3]  
[1, 2, 1, 2]  
[1, 2, 3, 4]
```

## **Complexity Analysis**

The complexity analysis is as follows:

**Time complexity:** Let the size of the paths list be  $p$  and the number of gardens is  $n$ . The number of flowers is fixed at 4. The loop for building the graph runs  $p$  times, giving a time complexity of  $O(p)$ . The outer loop for allocating types runs  $n$  times, and the inner loop for searching neighbors' type runs as many times as they are neighbors. As the number of the neighbors is limited to 3, we may consider the inner loop to have a time complexity of  $O(1)$ . The search within the set flowers also has the time complexity of  $O(1)$ . So, the overall time complexity is  $O(p+n)$ .

**Space complexity:** The data structure PathGraph contains  $n$  entries, each up to 3 elements. So, the space complexity is  $O(n)$ .

## **Conclusion**

We covered the basics of the graph theory and explained the representation and manipulation of graphs in Python. Then we covered four general questions that involved the formulation of word problems in terms of graphs. The questions that require traversing a graph in the BFS and the DFS manner will be covered in the next two chapters.

In the next chapter, we will be learning more about the depth-first search.

## **Points to remember**

- The graph is a collection of vertices and edges.

- A complete graph of  $n$  vertexes has  $n(n-1)/2$  edges.
- A connected acyclic graph is called a tree.
- In a tree,  $V = E + 1$
- The hash table is an array-like object where insertion and search can be done with  $O(1)$  time complexity. It is created by enclosing items in curly brackets.
- The dictionary is a convenient data structure to store graphs in an adjacency list form.
- The DFS and the BFS are methods of traversing a graph.

## MCQs

1. What is the number of edges in a complete graph having  $n$  nodes?
  - A.  $n(n-1)/2$
  - B.  $n(n+1)/2$
  - C.  $n^2$
  - D.  $\log n$
2. Which one of the below mentioned is not a linear data structure?
  - A. Queue
  - B. Stack
  - C. Arrays
  - D. Graph
3. A graph having  $n$  nodes will certainly have a parallel edge or a self-loop if:
  - A.  $E > n$
  - B.  $E > n^2$
  - C.  $E > n(n-1)/2$
  - D.  $E > n+1$
4. A connected acyclic graph is called:

- A. Regular graph
  - B. Tree
  - C. Forest
  - D. Path
5. The degrees of a pendant vertex and an isolated vertex are:
- A. 0 and 0
  - B. 0 and 1
  - C. 1 and 1
  - D. 1 and 0

## Answers to MCQs

**Q1:** A

**Q2:** D

**Q3:** C

**Q4:** B

**Q5:** D

## Questions

1. Describe two methods of representing a graph in memory.
2. A graph has 7 vertices; 3 of them have degree two, and 4 of them have degree one. Is this graph connected? (Hint:  $E = \frac{1}{2} \sum \deg(n) = 5$  and required  $E = V-1 = 6$ . There is one edge short, so the graph is not connected.)
3. A complete graph has  $n$  nodes. What is the degree of each node?
4. Describe two methods of traversing a graph.
5. Explain the terms simple graph, multigraph, and pseudo graph.

## Key terms

Set, dictionary, get, defaultdict, sort, graph, tree, node, edge, BFS, DFS, traversal.

# CHAPTER 5

## Depth First Search

Many practical problems can be modeled as the graph search problems. As noted in [Chapter 4](#), there are two methods for traversing a graph, namely Depth-First Search (DFS) and Breadth-First Search (BFS). In this chapter, we will study the problems that are amenable to DFS.

### Structure

We will first present the basic algorithm for traversing the graphs with DFS and Python code for implementing it. Then, we will take questions that use DFS modeling. We will cover the topics in the following order:

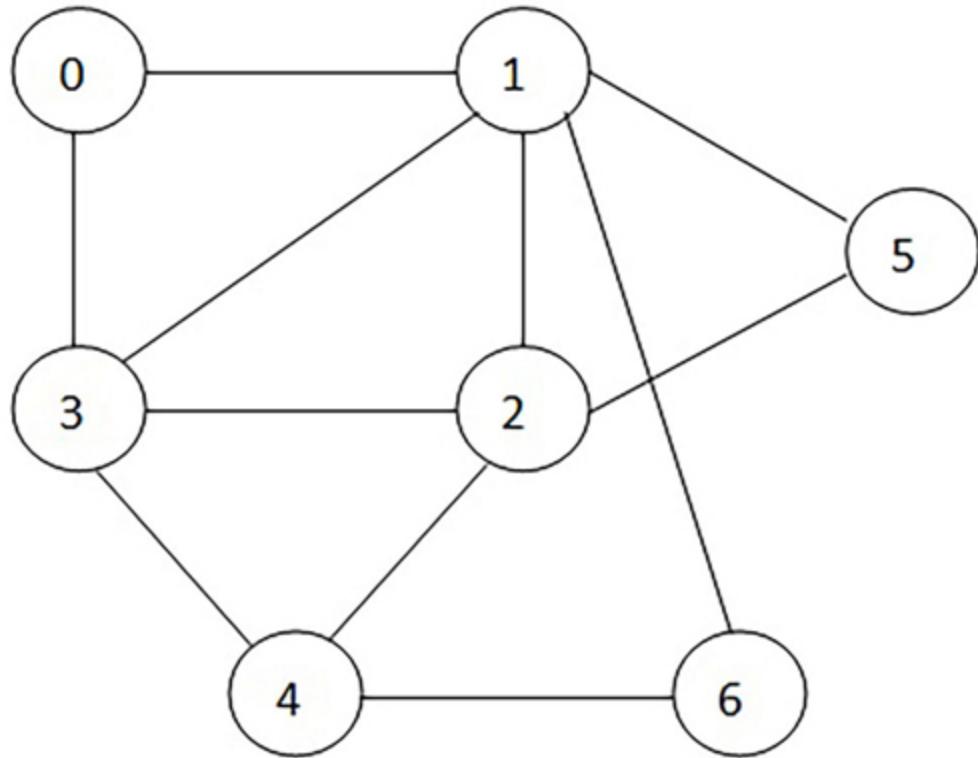
- DFS traversal of graphs and trees
- Question 36: How will you reconstruct the itinerary from tickets?
- Question 37: How will you validate a symmetric binary tree?
- Question 38: How will you find the maximum height of a binary tree?
- Question 39: How will you find path sum in a binary tree?
- Question 40: What is the  $K^{\text{th}}$  smallest element in a binary search tree?
- Question 41: How will you find the maximum path sum in a binary tree?
- Question 42: How will you validate a balanced binary tree?
- Question 43: How will you validate the binary search tree?
- Question 44: What is the number of islands?
- Question 45: How will you remove the surrounded islands?
- Question 46: Is the course schedule valid?
- Question 47: How will you reward a sales manager?

### Objectives

After studying this unit, you will be familiar with the DFS technique. You will be able to recognize problems that are suitable for the DFS modeling. You will be confident of solving any problem on the DFS.

## 5.1 DFS Traversal of Graphs and Trees

We will create a graph class having the functions of initialization, adding edges, and printing the nodes in the DFS order. Let us use the example graph of [Figure 5.1](#):



*Figure 5.1: Example graph*

We will first study the iterative approach which is easy to understand. The graph is stored in the adjacency list form. Referring to the methodology of section 5.1, the adjacency list for the graph of [Figure 5.1](#) is:

```
0:[1, 3]
1:[2, 3, 5, 6]
2:[3, 4, 5]
3:[4]
```

4 : [6]

For undirected graphs, in some problems, it is necessary to insert the destination vertex of an edge in the source vertex's adjacency list and insert the source vertex destination vertex's adjacency list. For our problem, this double insertion is not necessary. But if you do it, there is no harm. We have inserted the edges only once in the list mentioned earlier. The variables used in the program are as follows.

**g:** graph in the adjacency list form

**cur:** current node

**visited:** list of visited nodes

**st:** stack to store the nodes to be visited in the future

The DFS algorithm runs like this:

1. **Initialization:** First, we will create a list **visited** to store the visited nodes so that we don't traverse them again. Then, we will create a stack **st** to store the nodes to be visited in the future. The starting node is pushed on to the stack **st**. Any node may be arbitrarily chosen as the starting node.

Now, we will run a loop as long as the stack is not empty.

2. We will pop the top of the stack into the variable **cur**, denoting the current node. If **cur** is not present in the **visited** list, we will print it and add it to the **visited**.
3. Then we will scan the neighbors of **cur**, and if they are not present in **v**, we will add them on the stack.
4. If the stack is not empty, go to step 2. The loop will terminate when the stack becomes empty.

Let us walk through the example graph of [Figure 5.1](#). Initially, the stack contains the starting node 0, and **visited** is empty.

We will pop 0 into **cur** from the stack. Since it is not present in **visited**, we will print 0. Then we will scan the neighbors of 0, namely 1 and 3, and since they are not present in **visited**, we will push them onto the stack. The variables **visited** and **st** are:

```
v= [0], st = [1,3]
```

Now we will pop 3 from the stack. Since it is not present in visited, we will print 3. Then, we will scan the neighbors of 3, namely 4, and since it is not present in visited, we will push 4 onto the stack. The variables visited and st are:

```
v= [0,3], st = [1,4]
```

This process will continue till we finish all the elements in the stack. The movement of variables is shown in [Table 5.1](#):

Cur	Visited	St
0	[0]	[1,3]
3	[0,3]	[1,4]
4	[0,3,4]	[1,6]
6	[0,3,4,6]	[1]
1	[0,1,3,4,6]	[2,5]
5	[0,1,3,4,5,6]	[2]
2	[0,1,2,3,4,5,6]	[]

*Table 5.1: Movement of variables*

The output of the program is 0 3 4 6 1 5 2. Note that the order in which the neighbors are inserted in the stack in step 3 is arbitrary. Therefore, the sequence of a DFS search is not unique. We can make it unique if we employ some selection rules for choosing the order in which the neighbors are added to st. One commonly used rule is the alphabetical ordering of the node identifiers. This can be achieved by sorting the adjacency list. We will see this rule being applied in question 36.

The preceding algorithm prints only the nodes that are connected to the starting node. If the graph is disjoint, all the nodes will not be printed. To take care of this situation, we will choose an unvisited node as the starting node and repeat. We will keep repeating until there are no more unvisited nodes.

## [Python code](#)

The following code implements the above algorithm:

```
1. #DFS iterative approach
2. from collections import defaultdict
3. class Graph:
4.     def __init__(self):
5.         self.graph = defaultdict(list)
6.
7.     def insertEdge(self, v1, v2):
8.         self.graph[v1].append(v2)
9.     def DFS(self, startNode):
10.        visited = set()
11.        st = []
12.        st.append(startNode)
13.
14.        while(len(st)):
15.            cur = st.pop()
16.            if(cur not in visited):
17.                print(cur, end=" ")
18.                visited.add(cur)
19.
20.            for vertex in self.graph[cur]:
21.                if(vertex not in visited):
22.                    st.append(vertex)
23. #Driver code
24. g = Graph()
25. g.insertEdge(0,1)
26. g.insertEdge(0,3)
27. g.insertEdge(1,2)
28. g.insertEdge(1,3)
29. g.insertEdge(1,5)
30. g.insertEdge(1,6)
31. g.insertEdge(2,3)
32. g.insertEdge(2,4)
```

```
33. g.insertEdge(2,5)
34. g.insertEdge(3,4)
35. g.insertEdge(4,6)
36. g.DFS(0)
```

### **Output:**

```
0 3 4 6 1 5 2
```

Now we will study the recursive approach.

We need to create the `visited` list as a class variable because we don't want to get it initialized in each recursive call. We will create a non-recursive function, `DFS`, which initializes the `visited` list and calls the recursive function, `dfs`.

The algorithm is very simple. If the starting node is not in the visited list, we will enter it in the visited list and print it. After that, we will scan the neighbors list and pick up the first non-visited neighbor as the new starting node. Then, we will recursively call the `dfs`.

### **Python code**

The following code implements the recursive algorithm. The driver code and the class definition can be reused from the recursive example:

```
1. def DFS(self,node):
2.         self.visited = set() # Set to keep track of visited
nodes.
3.         self.dfs(node)
4.
5.     def dfs(self,node):
6.         if node not in self.visited:
7.             print (node,end=" ")
8.             self.visited.add(node)
9.             for neighbour in self.graph[node]:
10.                 self.dfs(neighbour)
```

### **Output:**

0 1 2 3 4 6 5

**Note:** The sequence is different from the iterative approach.

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If the number of vertices is  $V$ , and the number of edges is  $E$ , the time complexity is  $O(V+E)$ .

### **Space complexity:**

Due to the stack requirement, the space complexity is  $O(V)$ .

## DFS Traversal of Trees

There are three ways of traversing a tree using the DFS approach. They differ in their sequence of traversing a node and its children. Consider a node  $N$ , a left child  $L$ , and a right child  $R$ . In any method,  $L$  is traversed before  $R$ , but the sequence of  $L$ ,  $N$ , and  $R$  is different. The three ways are:

- In-order Traversal (Left-Node-Right)
- Pre-order Traversal (Node-Left-Right)
- Post-order Traversal (Left-Right-Node)

The time complexity of the traversal for all the three methods is  $O(n)$  because every node is visited exactly once. The time complexity using the BFS method is also  $O(n)$ .

The space complexity is different. For the BFS method, i.e. Level Order traversal, extra space is required, since the queue is proportional to the maximum width of the tree –  $w$ . Thus, the space complexity is  $O(w)$ . For the DFS method, the extra space required for the stack is proportional to the maximum height of the tree -  $h$ . Thus, the space complexity is  $O(h)$ .

The maximum possible width of a binary tree at height  $h$  is  $2^h$ , where the height of the root is 0. The total number of nodes,  $n$ , in a complete binary tree is  $2^{h+1} - 1$ . Consider the complete tree shown in [Figure 4.3 \(a\)](#). Its height is 2. At height 2 we have 4 nodes. The total number of nodes is

$1+2+4 = 7$ , which is  $2^{2+1} - 1$ . Thus, the maximum number of nodes of a tree is  $2^{h+1} - 1$ . Conversely, the minimum value of  $h$  for  $n$ -nodes of a binary tree is:

$$h = \log_2(n+1) - 1 = O(\log n)$$

On the other hand, the maximum height for  $n$  nodes can be  $n-1$ , when the tree is completely skewed, i.e., it contains only one node in each level. In this case, the tree resembles a singly-linked list.

The choice of method depends on the space complexity and the likely location of the object to be searched. If it is expected to be near the root, the BFS method will be quicker. If it is expected to be near the leaves, the DFS method will be quicker. During the DFS traversal, if we look for a back edge, it denotes the presence of a cycle. The DFS method is the best for detecting cycles.

If the nodes are unlabeled, then we can form  ${}^{2n}C_n/(n+1)$  binary trees from them. This is also equal to the number of binary search trees having  $n$  nodes. It is known as the Catalan number. If the nodes are labeled, then each combination can have  $n!$  permutations within it. Thus, the number of trees from  $n$  labeled nodes is  ${}^{2n}C_n/(n+1)*n!$ .

## 5.2 Question 36 – How will you reconstruct the itinerary from tickets?

This is an exercise of modeling a word problem as a graph. It teaches you to recognize the DFS situation in a given problem. We will also learn the use of a lambda function for sorting a structure.

### Problem statement

Your travel agent has given you a pile of air tickets but forgot to give the itinerary. You only know that you have to start from JFK (John F Kennedy airport, New York). Can you reconstruct the itinerary from the tickets? Note the following conditions:

1. A ticket consists of a source:destination pair, where the source and the destination are represented by 3-letter IATA codes. For example, JFK

for New York, BOM for Mumbai, NRT for Tokyo.

2. If there are multiple destinations for the same source, they must be traversed in the alphabetical order. For example, consider the tickets BOM:JFK and BOM:NRT. Then, BOM:JFK should be traversed first because alphabetically, JFK comes before NRT. There may be multiple tickets between the same source:destination pair.
3. All the tickets must be used, and used only once.

Let us illustrate with the help of two examples.

### **Example 1:**

**Input:** `[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`

**Output:** `["JFK", "MUC", "LHR", "SFO", "SJC"]`

**Explanation:** The starting is always JFK. Its destination is MUC. The destination of MUC is LHR. Thus, we will find the trail.

### **Example 2:**

**Input:** `[["JFK", "SFO"], ["JFK", "ATL"], ["SFO", "ATL"], ["ATL", "JFK"], ["ATL", "SFO"]]`

**Output:** `["JFK", "ATL", "JFK", "SFO", "ATL", "SFO"]`

**Explanation:** The starting is always JFK. From here we can go to SFO or ATL. If we choose SFO, the itinerary is:

`["JFK", "SFO", "ATL", "JFK", "ATL", "SFO"]`

But this is incorrect. When we have to choose between JFK:SFO and JFK:ATL, We will choose JFK:ATL, because ATL comes before SFO alphabetically.

### **Solution format:**

Your solution should be in the following format:

1. class Solution:
2.     def findItinerary(self, tickets: List[List[str]]) -> List[str]:

## Strategy

The given data is a list of the tickets. Each ticket is a source:destination pair. Thus, the ticket can be considered as a directional edge of a graph and the airport may be considered as a node. The source of the next ticket has to be the destination of the current ticket. This suggests that we have to traverse the graph in the DFS order. We have noted in section 5.1 that the DFS ordering is not unique. Our problem has one condition added to it that multiple destinations must be traversed in alphabetical order. This makes the answer unique. This condition can be easily met if we sort the ticket array based on the destination. Each element of the tickets array is an array itself. How do you sort such a structure? Let us introduce the lambda function for this. The `sort` function takes an optional argument `key` which is a function that tells how the sorting key will be generated for each element. We will use the lambda function to specify field 1 as the sorting key as follows:

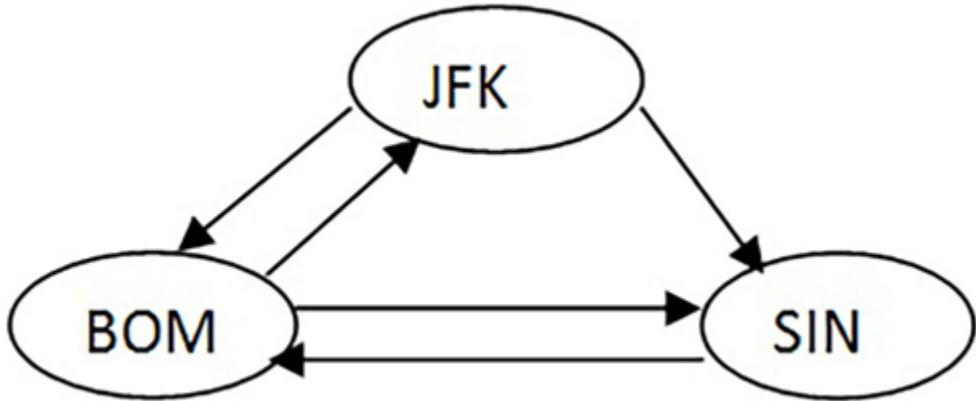
```
tickets.sort(key=lambda x:x[1])
```

But there could be more than one ticket between the same source:destination pair. So, we will modify the standard DFS algorithm that we saw in section 5.1. Instead of marking a node as visited, we will delete the traversed edge so that the ticket is not used again. Another variation is that we will not print the nodes as we traverse them. We will enter them in a class variable `path` and return it in the end.

We will illustrate the strategy with the help of the following example:

```
Tickets = [[“JFK”, “SIN”], [“JFK”, “BOM”], [“SIN”, “BOM”],  
[“BOM”, “JFK”], [“BOM”, “SIN”]]
```

The tickets can be represented as shown in [Figure 5.2](#):



**Figure 5.2:** Tickets graph

The first step is to read the tickets list and build a graph in the adjacency list format. It will be a dictionary where the key is the airport code and the value is the list of the destinations available for that airport. Initially, we will create an empty dictionary, named `dest`. We will scan the tickets array. If the source exists in the graph, we will add the destination to the destination list. If the source does not exist, we will create a new key and add a list containing destinations to it. At this stage, the graph looks like this:

```

BOM: ['JFK', 'SIN']
JFK: ['BOM', 'SIN']
SIN: ['BOM']

```

The second step is to traverse the graph making sure that all the tickets are used, and that a ticket is used only once. For traversing, we will write a recursive function, `dfs` (denoting Depth-First Search). This function takes in a source node as argument and does not return anything. Its output is appended to the class variable `path`. If the source node has outgoing edges, it first deletes that edge so that the ticket will not be reused. Then, it finds the destination for that source from the head of the destination list from the adjacency list and recursively calls itself passing the destination as the argument. After return from the recursive call, it appends the source node to the class variable `path`. As the innermost call appends the path first, the order of traversal, as seen from left to right is reversed. We could have avoided this problem by using `path.insert(0, s)` in place of `path.append(s)`, but this is inefficient. Instead of doing this, we will keep

appending the source at the end and reverse the array in the end. The function, `dfs`, gets called six times in the example. The status of variables through six calls is given in [Table 5.2](#):

Source	Destination graph	Destination
JFK	JFK:[BOM,SIN],SIN:[BOM],BOM:[JFK,SIN]	BOM
BOM	JFK:[SIN],SIN:[BOM],BOM:[JFK,SIN]	JFK
JFK	JFK:[SIN],SIN:[BOM],BOM:[SIN]	SIN
SIN	JFK:[],SIN:[BOM],BOM:[SIN]	BOM
BOM	JFK:[],SIN:[],BOM:[SIN]	SIN
SIN	JFK:[],SIN:[],BOM:[ ]	

*Table 5.2: Status of variables*

The recursion ends when there are no outgoing edges left. Thus the path is as follows:

```
[ 'JFK', 'BOM', 'JFK', 'SIN', 'BOM', 'SIN' ]
```

There is one more way in which the recursion can end. Consider the following example:

```
Tickets = [[ "JFK", "BOM" ], [ "BOM", "SIN" ], [ "SIN", "NRT" ]]
```

This results in a straight line itinerary like [ 'JFK', 'BOM', 'SIN', NRT ]. When we run `dfs("NRT")`, the destination dictionary does not have an entry for NRT because there was no outgoing edge from NRT. So, this is also a condition for terminating recursion.

## [Python code](#)

The following code implements the preceding algorithm:

```
1. from typing import List
2. class Solution:
3.     def findItinerary(self, tickets: List[List[str]]) ->
List[str]:
```

```

4.         self.dest = {} #Create empty graph of
source:destinations
5.         tickets.sort(key=lambda x:x[1]) #Sort tickets based
on destination
6.         for u,v in tickets:
7.             if u in self.dest: self.dest[u].append(v)#If u
exists, add v to list
8.             else: self.dest[u] = [v]     #u does not exist,
so create a new list
9.         self.path = []           #Create empty list for
storing path
10.        self.dfs("JFK")       #Start depth first search
11.        return self.path[::-1] #Reverse the order in
path
12.
13.    def dfs(self,s):
14.        #scan destination list of source s as long as it is
not empty
15.        while s in self.dest and len(self.dest[s]) > 0:
16.            v = self.dest[s][0]   #v = top of destination
list
17.            self.dest[s].pop(0) #Remove it from list (pop
from left)
18.            self.dfs(v)        #Recursively call v
19.            self.path.append(s) #Add s to path
20.
21. #Driver code
22. sol= Solution()
23. tickets =[["JFK","SIN"],["JFK","BOM"],["SIN","BOM"],["BOM",
"JFK"],["BOM","SIN"]]
24. print(tickets)
25. print(sol.findItinerary(tickets))

```

## **Output:**

```
[‘JFK’, ‘BOM’, ‘JFK’, ‘SIN’, ‘BOM’, ‘SIN’]
```

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** If the number of tickets is  $n$ , we are *calling dfs*  $(n+1)$  times. So, the time complexity is  $O(n)$ .

**Space complexity:** Due to the stack requirement, the space complexity is  $O(n)$ .

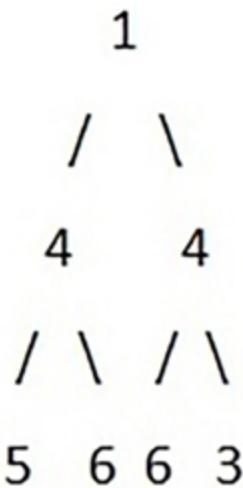
## 5.3 Question 37 – How will you validate a symmetric binary tree?

In this abstract problem, the DFS methodology has been applied.

### Problem statement

Find whether a given binary tree is symmetric, that is, it is a mirror image of itself.

For example, consider the binary tree [1, 4, 4, 5, 6, 6, 3] as shown in [Figure 5.3](#). (Please refer to section 4.3.1 for an array representation of the tree.) This tree is symmetric:



*Figure 5.3: Symmetric tree*

The trees [1, 2, 2, null, 3,null, 3] and [1, 2, 3, 5, 6, 5, 6] shown in [Figure 5.4](#) are not symmetric:



**Figure 5.4:** Asymmetric trees

We will use the `TreeNode` class introduced in section 4.3.1.

## Solution format

Your solution should be in the following format:

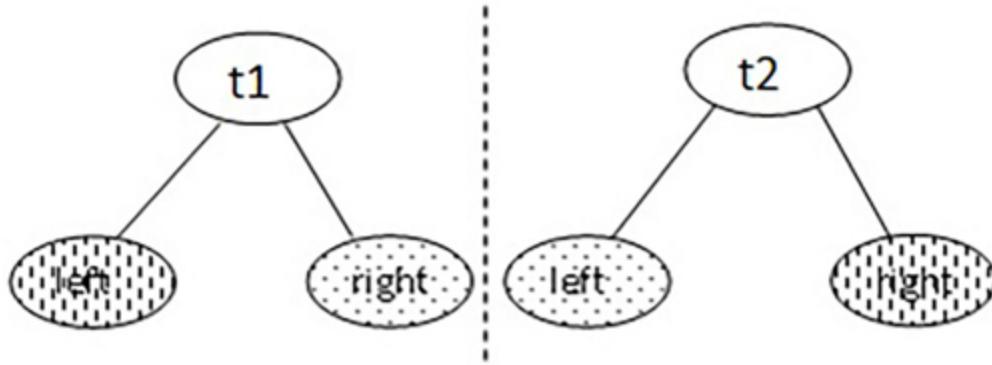
```
1. class Solution:  
2.     def isSymmetric(self, root: TreeNode) -> bool:
```

## Strategy

We will present a recursive approach and an iterative approach to solve the problem.

## Approach 1 – Recursive

Let us start from the root and test for the symmetry recursively. We will write a helper function `isMirror`, which takes in two tree roots (`t1` and `t2`) and determines if these are mirror images. To understand the test for symmetry, refer to [Figure 5.5](#):



**Figure 5.5: Symmetry condition**

For simplicity, assume that the children of  $t_1$  and  $t_2$  are the leaf nodes. Now  $t_1$  and  $t_2$  are mirror images if the following three conditions are met.

1.  $t_1.val = t_2.val$
2.  $t_1.left.val = t_2.right.val$
3.  $t_1.right.val = t_2.left.val$

If the children of  $t_1$  and  $t_2$  are not leaf nodes, then the function will be recursively called till we reach the leaf level.

The terminating conditions for the recursion are:

1. Both  $t_1$  and  $t_2$  are null, in which case, we return True.
2. One of the two is null, in which case, we return False.

## Python code

The following code implements this strategy:

```

1. # Definition for a binary tree node.
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class Solution:
9.     def isSymmetric(self, root: TreeNode) -> bool:

```

```

10.         return self.isMirror(root,root)
11.
12.     def isMirror(self,tree1,tree2):
13.         if(tree1 is None and tree2 is None):
14.             return True
15.         if(tree1 is None or tree2 is None):
16.             return False
17.         return (tree1.val==tree2.val) \
18.             and self.isMirror(tree1.right,tree2.left) \
19.             and self.isMirror(tree1.left,tree2.right)
20. Driver code
21. root = TreeNode(1)
22. root.left = TreeNode(4)
23. root.right = TreeNode(4)
24. root.left.left = TreeNode(5)root.left.right = TreeNode(6)
25. root.right.left = TreeNode(6)
26. root.right.right = TreeNode(5)
27. sol = Solution()
28. print(sol.isSymmetric(root))

```

## **Output:**

True

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

The number of recursive calls depends on the height of the tree, which can maximum be  $n$ . So, the space complexity is  $O(n)$ .

## **Approach 2 – Iterative**

We will create a queue of node pairs to be tested for symmetry. This technique is often used in the breadth-first search method. Initially, we will insert two copies of the root into the queue. Then, as long as the queue is not empty, we will pop two elements from the queue and examine them recursively. We will terminate the process when we reach the leaf nodes.

If the data value of the nodes in the pair is same and the children exist for both, we will insert two symmetrical pairs into the queue whose positions are shown in [Figure 5.5](#). The pairs are (left.left, right.right) and (left.right, right.left).

## Python code

The following code implements this strategy. The driver code is the same as earlier:

```
1. class Solution:
2.     def isSymmetric(self, root: TreeNode) -> bool:
3.         # Base case:if tree is empty it is considered
4.         # symmetric
5.         if (root == None) :return True
6.         # Base case:Single tree node with no children is
7.         # considered symmetric
8.         if(not root.left and not root.right):return True
9.         q = []      #Create empty queue of nodes
10.        q.append(root);q.append(root) #Add root to queue two
11.        times
12.        while(len(q)):
13.            # Pop first two nodes from queue
14.            # check their symmetry.
15.            leftNode = q[0] ; q.pop(0)
16.            rightNode = q[0] ;q.pop(0)
17.            l=leftNode.val;r=rightNode.val
```

```

17.          # if both left and right nodes exist but
different values
18.          # then return False
19.          if(leftNode.val != rightNode.val):return False
20.          #Left child of left node and right child of
right node
21.          # are symmetrical positions. Append them to
queue
22.          #Provided both exist
23.          if(leftNode.left and rightNode.right) :
24.              q.append(leftNode.left)
25.              q.append(rightNode.right)
26.          # If only one of the two children is
present
27.          # then tree is not symmetric.
28.          elif (leftNode.left or rightNode.right) :return
False
29.          #Right child of left node and left child of
right node
30.          # are symmetrical positions. Append them to
queue
31.          #Provided both exist
32.          if(leftNode.right and rightNode.left):
33.              q.append(leftNode.right)
34.              q.append(rightNode.left)
35.          # If only one of the two children is present
36.          # then tree is not symmetric.
37.          elif(leftNode.right or rightNode.left):return
False
38.          return True

```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

The length of the queue depends on the height of the tree, which can maximum be  $n$ . So, the space complexity is  $O(n)$ .

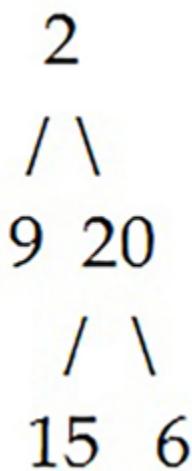
## **5.4 Question 38 – How will you find the maximum height of a binary tree?**

In this abstract problem, the DFS methodology has been applied.

### **Problem statement**

Find the maximum height of a given binary tree. The maximum height is equal to the number of nodes in the longest path from the root to the leaf nodes. (Note that this is because generally, we take the maximum height to be equal to the number of edges on the longest path from the root to the leaf nodes). You can use the `TreeNode` class defined in the previous example. Let us illustrate this with the help of an example.

#### **Example:**



*Figure 5.6: Example tree*

Consider the binary tree `[2, 9, 20, null, null, 15, 6]` as shown in [Figure 5.6](#). (Please refer to section 4.3.1 for the array representation of a tree.)

**Input:** root

**Output:** 3

**Explanation:** The longest paths are 2-20-6, and 2-20-15. So, the maximum height is 3.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def maxDepth(self, root: TreeNode) -> int:
```

## Strategy

Recursive calling is the most convenient here. Consider the root node 2 in [Figure 5.7](#). Its height can be found by adding 1 to the height of its children, namely 9 and 20, whichever is higher. We will write a recursive function `maxDepth` which takes in a parameter `root` and returns its height. For the base case (when `root` is `None`) 0, the height is returned. If the parameter `root` happens to be a leaf node, we return the height as 1, otherwise we will recursively call `maxDepth` to find the height of its left and right children and return the bigger among them as the height.

## Python code

The following code implements this strategy:

```
1. # Definition for a binary tree node.  
2. class TreeNode:  
3.     def __init__(self, v):  
4.         self.val = v  
5.         self.left = None  
6.         self.right = None  
7.  
8. class Solution:  
9.     def maxDepth(self, root: TreeNode) -> int:  
10.        if(root is None):           #Base case
```

```

11.         return 0
12.     if(root.left is None and root.right is None):
13.         return 1           #Leaf node
14.
15.         left = self.maxDepth(root.left)    #Recursively
call for height of left
16.         right = self.maxDepth(root.right) #Recursively call
for height of right
17.
18.         return max(left,right)+1        #Add 1 to the
maximum height of subtrees.
19. #Driver code
20. root = TreeNode(2)
21. root.left = TreeNode(9)
22. root.right = TreeNode(20)
23. root.right.left = TreeNode(15)
24. root.right.right = TreeNode(6)
25. sol = Solution()
26. print(sol.maxDepth(root))

```

## **Output:**

3

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

The length of the queue depends on the height of tree, which can maximum be  $n$ . So, the space complexity is  $O(n)$ .

## **5.5 Question 39 – How will you find the path sum in a binary tree?**

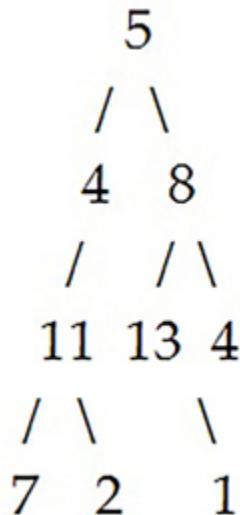
This is a variation of the DFS where you need to enumerate all the root-leaf paths and keep adding the node data.

## Problem statement

Find if the given binary tree has a root-to-leaf path such that the sum of all the node values along the path equals the given target. Recall that a leaf is a node with no children. Reuse the class `TreeNode` definition from the earlier problem. Let us illustrate this with the help of an example.

### **Example:**

Consider the binary tree in [Figure 5.7](#) and target = 22:



*Figure 5.7: Example figure*

**Input:** `root, 22`

**Output:** `True`

**Explanation:** We can find a root-to-leaf path 5-4-11-2, and its value sum is  $5+4+11+2=22$ . We should, therefore, return `True`.

## Solution format

Your solution should be in the following format:

1. `class Solution:`
2.     `def hasPathSum(self, root: TreeNode, sum: int) -> bool:`

## Strategy

We will use a recursive strategy. Our basic recurrence relation will be as follows:

Path sum of a node = Value of that node + Path sum of its child

The base case for recursion is reaching a leaf node.

We will define a helper function, `hasSum(self, root, sum, cur)`:

It is similar to the main function `hasPathSum`, except that it has one additional parameter `cur`, which represents the `sum` currently accumulated. We will add the root's value to `cur` and see if the target has been reached. If yes, and this is the leaf node, that is, the child nodes are null, we should return `True`. But before that, we need to see whether the current node is valid. It will be invalid when we try to descend below the leaf nodes. If both the child nodes are valid but the target has not been reached, it means that there is still further to go. So, we will recursively call `hasSum` with the left child and the right child as roots.

## Python code

The following code implements this strategy:

```
1. # Definition for a binary tree node.
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class Solution:
9.     def hasPathSum(self, root: TreeNode, sum: int) -> bool:
10.         return self.hasSum(root, sum, 0)
11.     def hasSum(self,root,sum,cur):
12.         if(root is None):
13.             return False
14.         if root.val == None:rv=0
```

```

15.         else:rv=root.val
16.         cur+=rv
17.         if(cur==sum and root.left is None and root.right is
None):
18.             return True
19.         return (self.hasSum(root.right,sum,cur) or
self.hasSum(root.left,sum,cur))
20. #Driver code
21. root = TreeNode(5)
22. root.left = TreeNode(4)
23. root.right = TreeNode(8)
24. root.left.left = TreeNode(11)
25. root.right.left = TreeNode(13)
26. root.right.right = TreeNode(4)
27. root.left.left.left = TreeNode(7)
28. root.left.left.right = TreeNode(2)
29. root.right.left.right = TreeNode(1)
30. sol = Solution()
31. print(sol.hasPathSum(root,27))
32. print(sol.hasPathSum(root,22))
33. print(sol.hasPathSum(root,17))

```

## **Output:**

True

True

True

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

The stack requirement depends on the height of the tree, which can maximum be  $n$ . So, the space complexity is  $O(n)$ .

## **5.6 Question 40 – What is the $K^{\text{th}}$ smallest element in a binary search tree?**

This is an application of the in-order traversing a binary search tree.

The binary search tree has a property that all the nodes in the left subtree of any given node are less than the given node, and all nodes in the right subtree of that given node are greater than the given node. In-order traversing is a special case of the DFS where the order of traversing is LNR, the left child, the node, and the right child. When a binary search tree is traversed by the in-order method, the nodes are visited in a sorted ascending order.

### **Problem statement**

A binary search tree is given in the form of a node list. Find the  $k^{\text{th}}$  smallest element in it. Let us illustrate this with the help of two examples.

#### **Example 1:**

Consider the binary tree of [Figure 5.8 \(a\)](#).

**Input:** root = [3, 1, 4, null, 2], k = 1.

**Output:** 1

**Explanation:** The node values in ascending order are (1,2,3,4). The first value is 1.

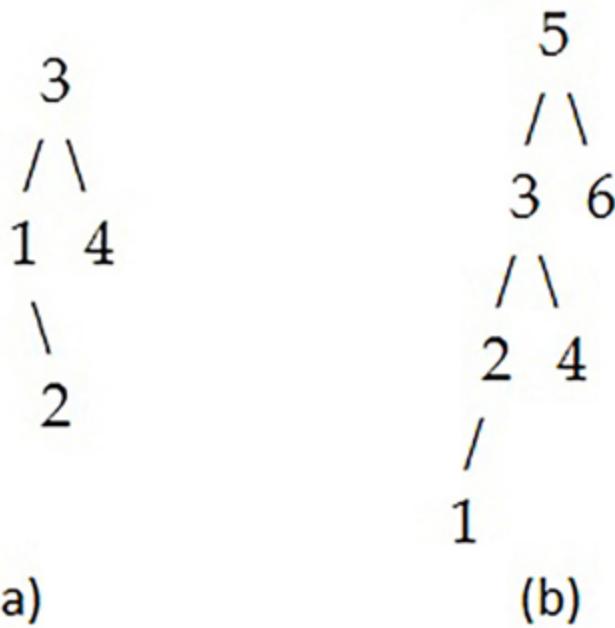
#### **Example 2:**

Consider the binary tree of [Figure 5.8 \(b\)](#).

**Input:** root = [5, 3, 6, 2, 4, null, null, 1], k = 3.

**Output:** 3

**Explanation:** The node values in ascending order are (1,2,3,4,5,6). The third value is 3.



*Figure 5.8: Example figure*

## Solution format

Your solution should be in the following format:

```

1. class Solution:
2.     def kthSmallest(self, root: TreeNode, k: int) -> int:

```

## Strategy

We will use the in-order traversing method on the given tree. As the nodes are traversed, we will enter them in an array, named sorted. Thus, the array will be in an ascending sorted order. The array index is zero based, so the  $k^{\text{th}}$  smallest element will be found at `sorted[k-1]`.

The `kthSmallest` function creates an empty array, named `sorted`, and calls the recursive function `printInorder(root)`. This function has to follow the LNR sequence of traversing. So, it first calls itself to get the left child. It keeps on calling itself till it hits a leaf node. It then starts the chain of returns. The first return that reaches the instruction `sorted.append(root.val)` comes from the deepest call. Then, it calls itself

to get the right child. The sequence continues till all invoked function calls eventually return.

## Python code

The following code implements this strategy. The actual code for the function `kthSmallest` is very small. Other parts are needed for testing it. The functions for the tree creation and printing and the driver code have been already explained in [Chapter 5](#). They can be reused here:

```
1. from typing import List
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class BinTree:
9.     def printTree(self, root:TreeNode)->None:
10.         LevelList = [root]
11.         self.printLevel(LevelList)
12.     def printLevel(self, LevelList:List[TreeNode])->
List[TreeNode]:
13.         LevelStr = ""
14.         outList = []
15.         ListEmpty = True
16.         for node in LevelList:
17.             if node is None:
18.                 LevelStr += "None"
19.                 outList.append(None)
20.                 outList.append(None)
21.             else:
22.                 LevelStr += (str(node.val) + " ")
23.                 outList.append(node.left)
24.                 outList.append(node.right)
```

```

25.                 ListEmpty = False
26.             if not ListEmpty:
27.                 print(LevelStr)
28.                 self.printLevel(outList)
29.
30. class Solution:
31.     def kthSmallest(self, root: TreeNode, k: int) -> int:
32.         sorted = []
33.         def printInorder(root):
34.             if root:
35.                 rv=root.val
36.                 printInorder(root.left) #Recursively call
left child
37.                 #print(root.val)
38.                 sorted.append(root.val) #Append node value
39.                 printInorder(root.right) #Recursively call
right child
40.             printInorder(root)
41.         return sorted[k-1]
42. #Driver code
43.
44. root = TreeNode(5)
45. root.left = TreeNode(3)
46. root.right = TreeNode(6)
47. root.left.left = TreeNode(2)
48. root.left.right = TreeNode(4)
49. root.right.left = TreeNode(None)
50. root.right.right = TreeNode(7)
51. bst = BinTree()
52. bst.printTree(root)
53. sol = Solution()
54. print("kth smallest=",sol.kthSmallest(root,2))

```

## **Output:**

```
5
3 6
2 4 None 7
kth smallest= 3
```

## Complexity Analysis:

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the recursion, we are using a stack of  $n$  elements and the sorted array of  $n$  elements. So, the space complexity is  $O(n)$ .

## 5.7 Question 41 – How will you find the maximum path sum in a binary tree?

This is an application of the DFS which requires a careful analysis of a node situation to decide the maximum path.

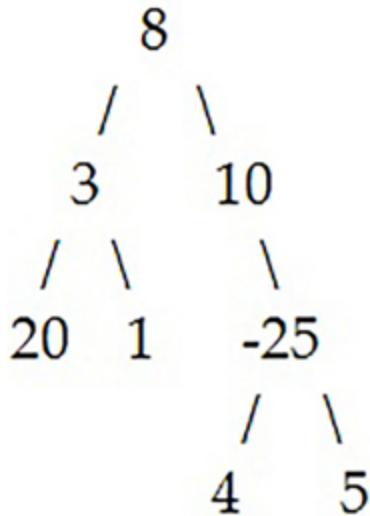
### Problem statement

Write a function to find the maximum path sum in a binary tree, where the path is a connected sequence of nodes and edges and the path sum means the sum of values of all the nodes in the path. The start and end nodes of a path can be any two nodes in the tree, that is, root, leaf, or intermediate. The node values may be positive or negative. Let us illustrate this with the help of an example.

### **Example:**

**Input:** [8, 3, 10, 20, 1, None, -25, None, None, None, None, None, None, 3, 4]

The tree is shown in [Figure 5.9](#):



**Figure 5.9:** Example figure

**Output:** 41

**Explanation:** The maximum sum is  $20+3+8+10 = 41$

## Solution format

Your solution should be in the following format:

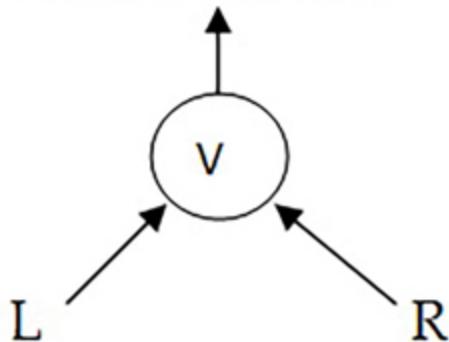
```

1. class Solution:
2.     def maxPathSum(self, root: TreeNode) -> int:
  
```

## Strategy

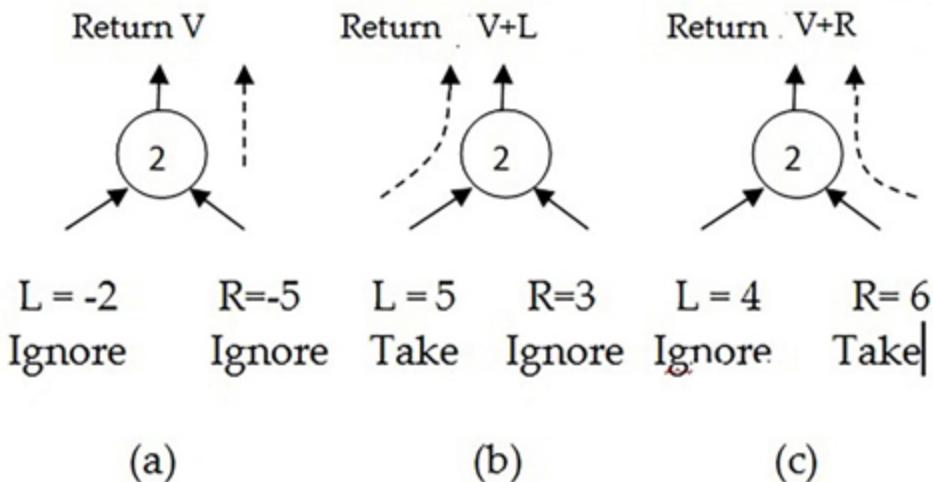
We will initialize a class variable, named `ans` to a large negative value. Ultimately, this will contain the answer. We will traverse through all the nodes and at each node, we will update `ans` whenever we find a higher value at the node. We will use a recursive function, named `solution` which evaluates the maximum possible contribution to the path sum up to that node. The situation of a node is shown in [Figure 5.10](#):

## Return value to caller



*Figure 5.10: Node situation*

The node receives maximum values from the left and right subtrees and its own value is  $V$ . Based on these three inputs, we have to decide what value is to be returned to the calling function. There can be three types of situations as shown in [Figure 5.11 \(a\)](#) through [Figure 5.11 \(c\)](#):



*Figure 5.11: Node decisions*

The calculation of the return values for each case is as follows:

- The values returned by the left and right sub trees ( $L$  and  $R$ ) are negative. There is no point in considering these sub trees in the maximum path. The maximum path (shown by the dotted arrow) should start from the node. The returned value will be  $V$ , that is, 2.
- The left sub tree returns a greater value (5). So, it is better to include it in the maximum path (shown by the dotted arrow). The returned value

will be  $V+L$ , that is,  $2+5=7$ .

- c. The right sub tree returns a greater value (6). So, it is better to include it in the maximum path (shown by the dotted arrow). The returned value will be  $V+L$ , that is,  $2+6=8$ .

The following statement calculates the return value `mxSide` according to the above rules. Note that `nv=V`, `left=L`, and `right=R`:

```
mxSide = max(nv, max(left, right)+nv)
```

In addition, we should also consider a possibility that our node may be the top node (root node) of the maximum path. In this case, the maximum path will go through the left child, the node, and the right child. The value addition of this path will be  $(L+V+R)$ . We will update the answer with the greater of `mxSide` and  $L+V+R$ .

## Python code

The following code implements this strategy. The actual code for the function `maxPathSum` is very small. Other parts are needed for testing it. The functions for tree creation and printing and the driver code have been already explained in [Chapter 5](#). They can be reused here:

```
1. from typing import List
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class BinTree:
9.     def printTree(self, root:TreeNode)->None:
10.         LevelList = [root]
11.         self.printLevel(LevelList)
12.     def printLevel(self, LevelList:List[TreeNode])->
List[TreeNode]:
13.         LevelStr = ""
```

```
14.         outList = []
15.         ListEmpty = True
16.         for node in LevelList:
17.             if node is None:
18.                 LevelStr += "None"
19.                 outList.append(None)
20.                 outList.append(None)
21.             else:
22.                 LevelStr += (str(node.val) + " ")
23.                 outList.append(node.left)
24.                 outList.append(node.right)
25.                 ListEmpty = False
26.             if not ListEmpty:
27.                 print(LevelStr)
28.                 self.printLevel(outList)
29.
30. class Solution:
31.     ans = -float("inf")
32.     def solution(self, node):
33.         #v=node.val if node is not None else 0
34.         if(node is None):
35.             return 0
36.         left = self.solution(node.left)
37.         right = self.solution(node.right)
38.         nv = node.val if node.val is not None else 0
39.         mxSide = max(nv,max(left,right)+nv)
40.         mxTop = max(mxSide, left+right+nv)
41.         self.ans = max(self.ans,mxTop)
42.         return mxSide
43.
44.     def maxPathSum(self, root: TreeNode) -> int:
45.         self.solution(root)
46.         return self.ans
47. #Driver code
```

```
48. root = TreeNode(8)
49. root.left = TreeNode(3)
50. root.right = TreeNode(10)
51. root.left.left = TreeNode(20)
52. root.left.right = TreeNode(1)
53. root.right.right = TreeNode(-25)
54. root.right.right.left = TreeNode(4)
55. root.right.right.right = TreeNode(5)
56.
57. bst = BinTree()
58. bst.printTree(root)
59. sol = Solution()
60. print("Max path sum=",sol.maxPathSum(root))
```

### **Output:**

```
8
3 10
20 1 None -25
None None None None None None 45
Max path sum= 41
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the recursion, we are using a stack of  $n$  elements. So, the space complexity is  $O(n)$ .

## 5.8 Question 42 – How will you validate a balanced binary tree?

The concept of a balanced binary tree is very important in determining the efficiency of the search operation in a binary search tree. The time complexity of searching is  $O(\log n)$  only if the binary search tree is balanced. As the imbalance increases, the time complexity leads towards  $O(n)$ .

A binary tree is said to be balanced if for every node, the difference between the heights of its left and right sub trees does not exceed 1. This is an extension of question no. 38, “the maximum height of a binary tree”.

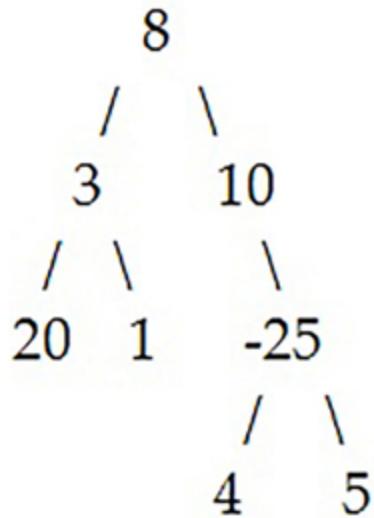
## Problem statement

Write a function to find if the given tree is balanced. Let us illustrate this with the help of two examples.

### **Example 1:**

Input: [8,3,10,20, 1, None, -25,None, None, None, None, None, None,4,5]

The tree is shown in [Figure 5.12](#):



*Figure 5.12: Example figure*

### **Output: False**

**Explanation:** The height of the left sub tree (3) is 2, and the height of the right sub tree (10) is 3. The difference (3-2) is within the permissible limits, but the tree is unbalanced because the balance has to be observed at every node. We notice that for node 10, the height of the left sub tree is 0, and the

height of the right sub tree (-25) is 2. As the difference of heights exceeds 1, the tree becomes imbalanced.

### Example 2:

**Input:** [8, 3, 10, 20, 1, None, -25]

**Output:** True

**Explanation:** This is the same as the preceding example, except that the last level is removed.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def isBalanced(self, root: TreeNode) -> bool:
```

### Strategy

We will define a nested recursive function, named `depth` which returns the height of the tree beginning at the root. It calls itself to find the depths of the left and the right sub trees. If the difference between the heights is less than 2, it returns the maximum of the two heights, otherwise it returns -1, to indicate the imbalance. If any of the sub tree heights is -1, it returns -1. If the final returned value is not -1, we will return True, otherwise we will return False.

### Python code

The following code implements this strategy. The actual code for the function `isBalanced` is very small. Other parts are needed for testing it. The functions for tree creation and printing, and the driver code have been already explained in [Chapter 4](#). They can be reused here:

```
1. from typing import List  
2. class TreeNode:  
3.     def __init__(self, v):  
4.         self.val = v  
5.         self.left = None
```

```

6.         self.right = None
7.
8. class BinTree:
9.     def printTree(self,root:TreeNode)->None:
10.        LevelList = [root]
11.        self.printLevel(LevelList)
12.    def printLevel(self,LevelList:List[TreeNode])->
List[TreeNode]:
13.        LevelStr = ""
14.        outList = []
15.        ListEmpty = True
16.        for node in LevelList:
17.            if node is None:
18.                LevelStr += "None "
19.                outList.append(None)
20.                outList.append(None)
21.            else:
22.                LevelStr += (str(node.val) + " ")
23.                outList.append(node.left)
24.                outList.append(node.right)
25.                ListEmpty = False
26.        if not ListEmpty:
27.            print(LevelStr)
28.            self.printLevel(outList)
29. class Solution:
30.     def isBalanced(self, root: TreeNode) -> bool:
31.         #Nested function to find depth. (-1) denotes
unbalanced
32.         def depth(root: TreeNode) -> int:
33.             rv = root.val if root else 0
34.             if not root: return 0          #Depth is 0 if node
is None
35.             ldepth = depth(root.left)   #Find depth of left
subtree

```

```

36.                 rdepth = depth(root.right) #Find depth of
right subtree
37.                 if ldepth == -1 or rdepth == -1 or abs(ldepth -
rdepth) > 1:
38.                     return -1
39.                 return max(ldepth, rdepth) + 1 #Depth of root =
sub trees+1
40.             dr = depth(root)
41.             return dr != -1
42. #Driver code
43. root = TreeNode(8)
44. root.left = TreeNode(3)
45. root.right = TreeNode(10)
46. root.left.left = TreeNode(20)
47. root.left.right = TreeNode(1)
48. root.right.right = TreeNode(-25)
49. root.right.right.left = TreeNode(4)
50. root.right.right.right = TreeNode(5)
51.
52. bst = BinTree()
53. bst.printTree(root)
54. sol = Solution()
55. print("Tree is balanced?", sol.isBalanced(root))

```

## **Output:**

```

8
3 10
20 1 None -25
None None None None None None 4 5
Tree is balanced? False

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the recursion, we are using a stack of  $n$  elements. So, the space complexity is  $O(n)$ .

## **5.9 Question 43 – How will you validate a binary search tree?**

The concept of a binary search tree (BST) is very useful for fast searching. But we need to test whether the given BST is valid or not. A BST is valid if the following conditions are met for every node.

1. The left subtree of a node contains only the nodes with the keys less than the node's key.
2. The right subtree of a node contains only the nodes with the keys greater than the node's key.
3. Both the left and the right subtrees must also be the binary search trees.

### **Problem statement**

Write a function to find if the given binary tree is a valid binary search tree. Let us illustrate this with the help of two examples.

#### **Example 1:**

**Input:** Tree = [2, 1, 3]

The tree is shown in [Figure 5.13 \(a\)](#).

**Output:** True

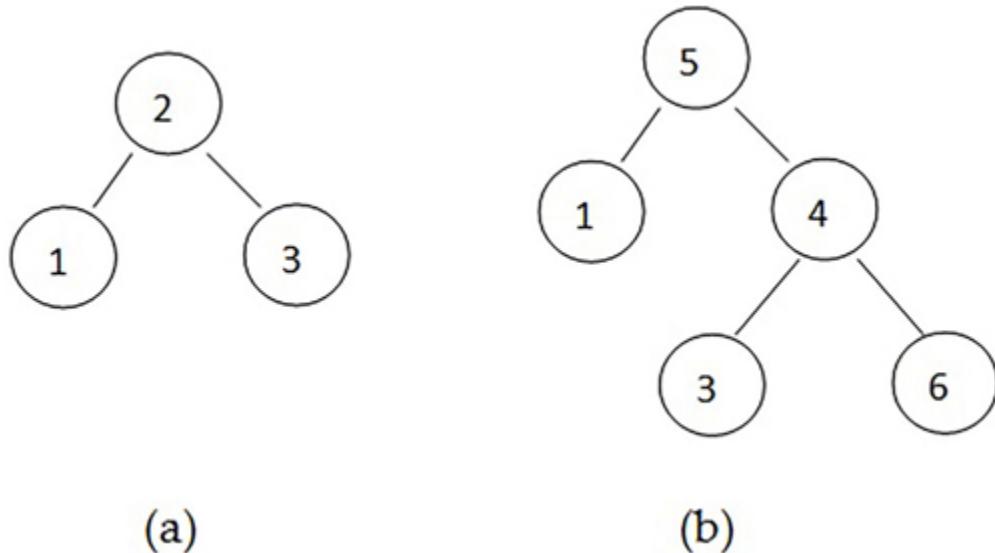
**Explanation:** For node 2, the left child is less than 2, and the right the right child is greater than 2.

#### **Example 2:**

**Input:** Tree = [5, 1, 4, null, null, 3, 6]

The tree is shown in [Figure 5.13 \(b\)](#).

**Output:** False



*Figure 5.13: Example figure*

**Explanation:** For node 5, the right child of 5 is 4, which is not allowed.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def isValidBST(self, root: TreeNode) -> bool:
```

## Strategy

At every node we need to ascertain that the left value is less than the value of its parent, and the right value is greater than the value of its parent. So we will create a variable, `lower`, which acts as the limit for the values in the left subtree. Similarly, the `upper` defines the limit for values in the right subtree. We will define a nested recursive function, named `helper`. It takes in three arguments, namely the node, the `lower`, and the `upper`. It returns a Boolean value `True` if the tree is valid up to the current node. It first checks the base case of `node = None`. If it is `None`, it returns `True`. If the node is valid, it checks for the following condition:

```
lower < node.val < upper
```

If the above condition is not met, the function returns `False`. The default values of the lower and the upper are  $-\infty$  and  $+\infty$ . When the first call to the helper is made, the node is the root, and the `lower` and `upper` parameters are not passed. So, the default values are used.

It calls itself and passes the parameters `node.right`, `val`, and `upper`, that is, it descends to the right subtree and makes the current `val` as the new lower limit. It does not change the upper. It keeps on descending the right branch till it reaches a dead end, that is, the node is `None`.

After returning, it calls itself and passes the parameters `node.left`, `lower`, and `val`, that is, it descends to the left subtree and makes the current `val` as the new upper limit. It does not change the `lower`. It keeps on descending the left branch till it reaches a dead end, that is, the node is `None`.

By this time, if it has not encountered a `False` condition anywhere, it returns `True`.

The algorithm will be clear from the status of various variables at each call. Let the input be `[2, 1, 4, None, None, 3, 6]`. The statements labeled `#diagnostic` should be uncommented for getting the following output:

```
2
1 4
None None 3 6
val  left   right   lower   upper
2    1      4        -inf     inf
4    3      6        2       inf
6    0      0        4       inf
Node is None
Node is None
3    0      0        2       4
Node is None
Node is None
1    0      0        -inf     2
Node is None
Node is None
Tree is valid? True
```

## Python code

The following code implements this strategy. The actual code for the function `isValidBST` is very small. Other parts are needed for testing it. The functions for tree creation and printing, and the driver code have been already explained in [Chapter 4](#). They can be reused here:

```
1. from typing import List
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class BinTree:
9.     def printTree(self, root:TreeNode)->None:
10.         LevelList = [root]
11.         self.printLevel(LevelList)
12.     def printLevel(self, LevelList:List[TreeNode])->
List[TreeNode]:
13.         LevelStr = ""
14.         outList = []
15.         ListEmpty = True
16.         for node in LevelList:
17.             if node is None:
18.                 LevelStr += "None"
19.                 outList.append(None)
20.                 outList.append(None)
21.             else:
22.                 LevelStr += (str(node.val) + " ")
23.                 outList.append(node.left)
24.                 outList.append(node.right)
25.             ListEmpty = False
26.         if not ListEmpty:
27.             print(LevelStr)
```

```
28.             self.printLevel(outList)
29. class Solution:
30.     def isValidBST(self, root: TreeNode) -> bool:
31.         def helper(node, lower = float('-inf'), upper =
float('inf')):
32.
33.             if node is None:
34.                 print("Node is None")
#Diagnostic
35.             return True
36.             nl=node.left.val if node.left else
0 #Diagnostic
37.             nr=node.right.val if node.right else
0 #Diagnostic
38.             val =
node.val #Diagnostic
39.
print(val,nl,nr,lower,upper) #Diagnostic
40.             if not val: val = 0
41.             if val <= lower or val >= upper:
42.                 return False
43.
44.             if not helper(node.right, val, upper):
45.                 return False
46.             if not helper(node.left, lower, val):
47.                 return False
48.             return True
49.         return helper(root) #Driver code
50.
51. root = TreeNode(5) #Change this to 2
52. root.left = TreeNode(1)
53. root.right = TreeNode(4)
54. root.right.left = TreeNode(3)
55. root.right.right = TreeNode(6)
```

```
56. bst = BinTree()  
57. bst.printTree(root)  
58. sol = Solution()  
59. print("Tree is valid?",sol.isValidBST(root))
```

**Output:** With the diagnostic statements commented

```
5  
1 4  
None None 3 6  
Tree is valid? False
```

**Output:** With the diagnostic statements commented, and the root changed to 2

```
2  
1 4  
None None 3 6  
Tree is valid? True
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the recursion, we are using a stack of  $n$  elements. So, the space complexity is  $O(n)$ .

## 5.10 Question 44 – What is the number of islands?

Here we have to collect the similar adjacent elements in a group. When we read the word *adjacent* in the problem, we can guess that it belongs to the graph theory. However, this problem does not actually require a standard representation in the form of the adjacency list. It is more convenient to get the neighbors' list by the physical proximity. However, we will use the

standard DFS technique to pursue the neighbors of an element. This type of algorithm is useful in the digital image processing.

## **Problem statement**

You are given a matrix which represents a large lake. The lake has several islands in it. In the matrix, water is represented by 0, and land is represented by 1. An island is formed by connecting the adjacent lands horizontally or vertically. It is surrounded by water on all sides. The islands may touch the edges, that is, we may assume that there is water beyond the four edges of the grid. Let us illustrate this with the help of two examples.

### **Example 1:**

#### **Input:**

```
grid = [
    ["1", "1", "1", "1", "0"],
    ["1", "1", "0", "1", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "0", "0", "0"]
]
```

#### **Output:** 1

**Explanation:** There is a single connected piece consisting of  $(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 3), (2, 0), (2, 1)$ .

### **Example 2:**

```
Input: grid = [
    ["1", "1", "0", "0", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "0", "0", "1", "1"]
]
```

#### **Output:** 3

**Explanation:** There are three connected pieces.

1:  $(0, 0), (0, 1), (1, 0), (1, 1)$

```
2: (2, 2)
3: (3, 3), (3, 4)
```

## Solution format

Your solution should be in following format:

```
1. class Solution:
2.     def numIslands(self, grid: List[List[str]]) -> int:
```

## Strategy

We will consider each land element of the grid as a node in a graph. But instead of creating the graph in the standard adjacency list form, we will use the grid itself to get the neighbors of a node. We will start from a seed node and grow it in all the directions in a DFS manner. Instead of keeping a separate list of visited nodes, we will use the grid itself for this purpose. When we will traverse a node  $(i, j)$ , we will convert it into water by making the  $\text{grid}[i, j]=0$  so that it will not be considered again.

We will create a nested recursive function `set_zero` to discover and set the connected elements to zero. The base cases of the recursion will occur when the indices hit the edge or when the land is not found. Notice the statement:

```
if (0 <= row <= nrow-1)
```

This is a beautiful way of expressing that the row lies between 0 and `nrow-1` in Python. If the base case is not found, the function calls itself 4 times with the arguments as top, bottom, right, and left neighbors. As `set_zero` is a nested function, it is not allowed to write into the outer variables. Therefore, the `grid` is passed as an argument to `set_zero`.

To begin with, we will estimate the dimensions of the grid. The `grid` is presented as a list of lists. The number of rows, `nrow` is obtained as `len(grid)`. `grid[0]` represents the first row of the grid. The number of columns, `ncol` is obtained as `len(grid[0])`.

Then we will run a nested loop in `i` and `j` to scan the grid. Whenever we find a piece of land, we will call `set_zero` function to explore the whole

island and sink it. On return from set\_zero, we will increment the island count.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def numIslands(self, grid: List[List[str]]) -> int:
4.         if len(grid) == 0: return 0 #Base case
5.
6.         n_islands = 0
7.         nrow = len(grid)
8.         ncol = len(grid[0])
9.         #Recursive nested function to wipe out neighbours of
10.        (row,col)
11.        def set_zero(row, col, grid):
12.            if (0 <= row <= nrow-1) and (0 <= col <= ncol-1):
13.                if grid[row][col] == '1':
14.                    grid[row][col] = '0'
15.                    set_zero(row+1, col, grid)
16.                    set_zero(row-1, col, grid)
17.                    set_zero(row, col+1, grid)
18.                    set_zero(row, col-1, grid)
19.            #Scan the matrix for 1 value
20.            for row in range(nrow):
21.                for col in range(ncol):
22.                    if grid[row][col] == '1':
23.                        n_islands += 1
24.                        set_zero(row, col, grid)
25.        return n_islands
26. #Driver code
```

```

27. sol = Solution()
28. grid = [
29.     ["1","1","1","1","0"],
30.     ["1","1","0","1","0"],
31.     ["1","1","0","0","0"],
32.     ["0","0","0","0","0"]
33. ]
34. print("Number of islands = ",sol.numIslands(grid))
35. grid = [
36.     ["1","1","0","0","0"],
37.     ["1","1","0","0","0"],
38.     ["0","0","1","0","0"],
39.     ["0","0","0","1","1"]
40. ]
41. print("Number of islands = ",sol.numIslands(grid))

```

### **Output:**

Number of islands = 1  
 Number of islands = 3

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Eventually, the number of elements scanned is  $nrows * ncol$ . If  $nrows = ncol = n$ , then, the overall time complexity is  $O(n^2)$ .

### **Space complexity:**

During the recursion, we may go  $n^2$  deep in the worst case. So, the space complexity is  $O(n^2)$ .

## **5.11 Question 45 – How will you remove the surrounded islands?**

This question is similar to the previous one. Here, we want to remove an island if it is surrounded by water on all sides, that is, it does not touch an edge. It can be modeled with the graph theory and like the previous solution, we will not use the adjacency list form here as well.

## **Problem statement**

You are given a matrix named `Board` which represents a rectangular lake. Water is represented by the letter ‘X’ and land is represented by the letter ‘O’. Write a function to remove an island (that is, convert it into water) if it is surrounded by water on all sides. In other words, any ‘O’ that does not lie on the border and is not connected to any other ‘O’ on the border will be converted into ‘X’. Two cells are said to be connected if they are adjacent horizontally or vertically. Let us illustrate this with an example.

### **Input:**

```
Board = [['X', 'X', 'X', 'X'],
          ['X', 'O', 'O', 'X'],
          ['X', 'X', 'O', 'X'],
          ['X', 'O', 'X', 'X']]
```

### **Output:**

```
Board = [['X', 'X', 'X', 'X'],
          ['X', 'X', 'X', 'X'],
          ['X', 'X', 'X', 'X'],
          ['X', 'O', 'X', 'X']]
```

**Explanation:** The island formed by the pixels [1,1], [1,2], and [2,2] will be removed as it is surrounded by water on all sides. The pixel [3,1] will remain intact because it lies on the border.

## **Solution format**

Your solution should be in the following format:

1. class Solution:
2.     def solve(self, board: List[List[str]]) -> None:

## Strategy

We will develop a simple heuristic strategy. We know that the pixels touching the border are safe. So, firstly, we will convert all the border pixels and their connections to an intermediate value ‘P’ (meaning protected) using a helper function `dfs`. The surrounded ‘O’ pixels cannot be reached, so they will remain ‘O’. Then, we will scan the matrix and convert these ‘O’ pixels into ‘X’, and convert ‘P’ pixels back to ‘O’.

We will write a nested function `dfs(i, j)`, which recursively searches the left, right, top, and bottom neighbors of the `board[i][j]` for ‘O’ and converts them into ‘P’. The recursion is terminated when `i` and `j` hit the limits.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def solve(self, board: List[List[str]]) -> None:
4.         #Base cases
5.         if not board: return None
6.         if len(board)==1 or len(board)==2 \
7.             or len(board[0])==1 or len(board[0])==2:
8.             return None
9.         #Function Converts all neighbours of i,j from 0 to P
10.        def dfs(board,i,j):
11.            if i<0 or j<0 or i>len(board)-1 \
12.                or j>len(board[0])-1 or board[i][j]=='X':
13.                return
14.            if board[i][j]=='0':
15.                board[i][j]='P'
16.                dfs(board,i+1,j)
17.                dfs(board,i-1,j)
18.                dfs(board,i,j+1)
19.                dfs(board,i,j-1)
```

```

20.     #Convert all 0's connected to top and bottom row to P
21.     for i in range(len(board[0])):
22.         if board[0][i]=='0': dfs(board,0,i)
23.         if board[len(board)-1]
24.             [i]=='0':dfs(board,len(board)-1,i)
24.     #Convert all 0's connected to top and bottom col to P
25.     for j in range(len(board)):
26.         if board[j][0]=='0': dfs(board,j,0)
27.         if board[j][len(board[0])-1]=='0':
28.             dfs(board,j,len(board[0])-1)
29.     #Scan the whole matrix, convert P to 0, 0 to X
30.     for i in range(len(board)):
31.         for j in range(len(board[0])):
32.             if board[i][j]=='0': board[i][j]='X'
33.             if board[i][j]=='P': board[i][j]='0'
34. #Driver code
35. b = [['X','X','X','X'],
36.       ['X','0','0','X'],
37.       ['X','X','0','X'],
38.       ['X','0','X','X']]
39. sol=Solution()
40. sol.solve(b)
41. for i in range(len(b)): print(b[i])

```

### **Output:**

```

['X', 'X', 'X', 'X']
['X', 'X', 'X', 'X']
['X', 'X', 'X', 'X']
['X', '0', 'X', 'X']

```

### **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

Each pixel is covered 3 times and the number of pixels is  $n^2$ . So, the time complexity is  $O(n^2)$ .

### **Space complexity:**

During the recursion, we may go  $n^2$  deep in worst case. So, the space complexity is  $O(n^2)$ .

## **5.12 Question 46 – Is the course schedule valid?**

Here is a practical application of the DFS technique.

### **Problem statement**

A university runs various courses that have to be taken in a particular sequence. Every course  $c$  has a pre-requisite  $p$ . For convenience, let us assume that the course codes  $c$  and  $p$  are integers. The dependency relationship is described as a pair  $[c, p]$ . You are given a list of the dependency relationships. Write a function to test if you can complete all the courses. Return `True` if the course schedule is valid, else return `False`. Let us illustrate this with the help of an example.

### **Example:**

**Input:** Prerequisites = `[[1, 0], [2, 0], [3, 1], [3, 2]]`

**Output:** `True`

**Explanation:** 0 should be done before 1 and 2. 1 or 2 should be done before 3. These conditions are satisfied either by `[0,1,2,3]` or by `[0,2,1,3]`. So return `True`.

### **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def canFinish(self, numCourses: int, prerequisites:  
List[List[int]]) -> bool:
```

### **Strategy**

The moment you see the dependency relationship, you can immediately think that this is a graph theory problem where the relation  $[c, p]$  represents a directed edge, and  $c, p$  are the nodes of the graph. Since we need to find the sequence of the courses, the DFS is a natural choice. Now let us think why we will not be able to complete all courses? Consider the relations  $[c_1, c_2], [c_2, c_1]$  which dictate that the course  $c_2$  must be done before the course  $c_1$ , and the course  $c_1$  must be done before the course  $c_2$ . This is impossible. Or let us consider a longer relation list  $[c_4, c_3], [c_3, c_2], [c_2, c_1], [c_1, c_4]$ . The first three conditions imply the sequence  $c_1-c_2-c_3-c_4$ . But the last condition requires that  $c_4$  should be done before  $c_1$ . Again this is impossible. So, now we can state formally that for the course schedule to be valid, there should be no cycles in the graph.

In *Question 32* in [Chapter 4](#), we found the redundant connection by finding cycles. We used the union and find approach there. But that was an undirected graph. We cannot use this approach in a directed graph. This is because a directed graph cannot be represented using the disjoint-set (the data structure on which the union-find is performed). When we say ‘ $a$  union  $b$ ’, we cannot make out the direction of the edge:

1. Is  $a$  going to  $b$ ? (or)
2. Is  $b$  going to  $a$ ?

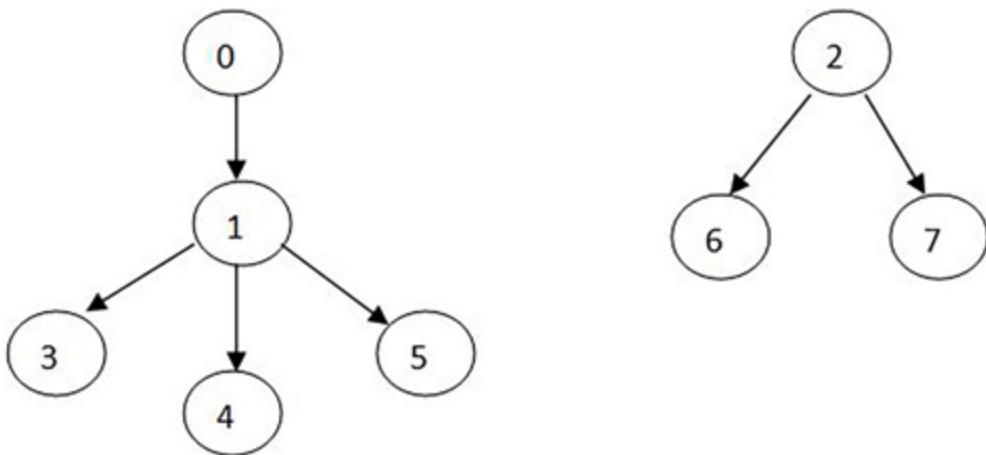
We will use the DFS approach for solving this problem. First, let us discuss the methodology, and then we will come to the code. Let the edges list be:

```
prerequisites = [[0,1],[1,3],[1,4],[1,5],[2,6],[2,7]]
```

The adjacency list is as follows:

```
0: [1]
1: [3, 4, 5]
2: [6, 7]
```

The corresponding graph is shown in [Figure 5.14](#):



*Figure 5.14: Example graph*

From the adjacency list, we observe that there are total 8 nodes out of which there are 3 nodes having the outgoing edges, namely 0, 1, and 2. We want to find which nodes we can visit, starting from these nodes. We will create a set of 8 elements named `color` that stores the visited status of the nodes. We will have three states of visiting:

1. **White**: The node has not been visited.
2. **Gray**: The node has been visited as a neighbor.
3. **Black**: The node is explored, that is, all the neighbors of this node have been visited.

This extra mark is necessary because you may visit this node in the next search. If it remains gray, it will be erroneously treated as visited in the search, and a `false cycle` will be detected. If it is marked black, it will be skipped.

Initially, all the nodes will be white. Then, we will scan the course list and if the color of a course is found to be white, we will call the recursive function `dfs_visit` to traverse its neighbors. The neighbors will be painted gray when they are visited. During this process, if we find a node that is already gray, then we have found a cycle. When all the neighbors are done, we will paint u black.

Now let us walk through the code. First of all, our code will build the adjacency list from the list of edges. It is stored in the form of a dictionary, named `G`. The use of the `defaultdict` class makes it very easy. Then, we

will create a set named `color` to store the color of the nodes. We will also create a Boolean variable `found_cycle` and initialize it to `False`. These are all class variables so that they can be accessed from the helper function `dfs_visit`.

We will run a main loop with the index `u` that spans all the entries in `color`. If we find that the `color[u]` is white, we will call `dfs_visit(u)`. This function checks if the `found_cycle` is true, and if it is true, it returns. Then it paints `u` as gray (visited as neighbor). Then it scans the list of neighbors of `u`. If it finds a neighbor painted gray, it makes the `found_cycle` true. If the neighbor is white (unvisited), it calls itself with the neighbor as argument to descend one level of depth. When the list of the neighbors is finished, or when there are no neighbors left, the node `u` is painted black (explored).

We will start the main loop with `u = 0`. The function `dfs_visit` is recursively called for the nodes 0, 1, 3, 4, and 5, and they will all be painted black. When `u` becomes 1 in the next iteration, the call to `dfs_visit` is skipped because the `color[1]` is not white. When `u` becomes 2, the `dfs_visit` is called and the nodes 2, 6, and 7 turn black. In this process, we do not find a cycle, so we will return `True`, that is, the schedule is valid.

If you add an edge [3, 0], then the cycle will be formed.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. from collections import defaultdict
3. class Solution:
4.     def canFinish(self, numCourses: int, prerequisites:
List[List[int]]) -> bool:
5.         G = defaultdict(list)
6.         for i, j in prerequisites:
7.             G[i].append(j)
8.         self.color = {u : "white" for u in
range(numCourses) }
```

```

9.         self.found_cycle = False # - found_cycle is
initially false
10.        for u in range(numCourses):# - Visit all nodes.
11.            #c=self.color #Added for debugging
12.            if self.color[u] == "white":
13.                self.dfs_visit(G, u)
14.            if self.found_cycle:
15.                break
16.        return not self.found_cycle
17.    def dfs_visit(self,G, u):
18.        #c=self.color #Added for debugging
19.        if self.found_cycle:return #End of recursion
20.        self.color[u] = "gray" # - Gray nodes are in the
current path
21.        for v in G[u]: # - Check neighbors,in adjacency
list
22.
23.            if self.color[v] == "gray": # If neighbor is
gray, there is loop
24.                self.found_cycle = True
25.                return
26.            if self.color[v] == "white": # call dfs_visit
recursively.
27.                self.dfs_visit(G, v)
28.            self.color[u] = "black" #All neighbors visited
29. sol=Solution()
30. pre=[[0,1],[1,3],[1,4],[1,5],[2,6],[2,7]]
31. print(sol.canFinish(8,pre))

```

**Output:** True

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:**

If  $V$  is the number of courses, and  $E$  is the number of dependencies, then, the time complexity is  $O(V+E)$ .

### Space complexity:

The space complexity is  $O(V+E)$ .

## 5.13 Question 47 – How will you reward a sales manager?

Suppose, you are the CEO of a multinational company. Your sales offices are spread across many countries. Each country has several zonal offices and each zone has several city offices. For example, India may have four zones, namely East, West, North, and South. The west zone has city offices in Mumbai, Pune, and Nagpur. The hierarchy may continue like this, but let us stop at the city level. There are salesmen in each city. You want to reward the salesmen in proportion to the sales achieved by them. Also, you want to reward the city manager in proportion to the sales achieved in his city, the zone manager in proportion to the sales achieved in his zone, the country manager in proportion to the sales achieved in his country, and the global sales manager in proportion to the sales achieved worldwide. Note that the managers do not simply sit in their cabins. They also handle some important customers and they have their own sales. We need to write a program to find the total sales achieved by any employee along with his team.

### Problem statement

The hierarchy of a company's sales force is represented in a tree structure. Each node of the tree is an object of the following class:

```
1. class Employee:  
2.     def __init__(self, id: int, sales: int, subordinates:  
List[int]):  
3.         self.id = id                      #Employee ID  
4.         self.sales = sales                 #Sales achieved  
5.         self.subordinates = subordinates   #List of  
subordinate
```

You are given the employee information in the form of a list of employee objects and an ID. Find out how much sales were achieved by this employee along with his subordinates. Let us illustrate this with the help of an example.

**Example:**

**Input:** The employee data in the form of a list of records [id, sales, id list of subordinates] is as follows:

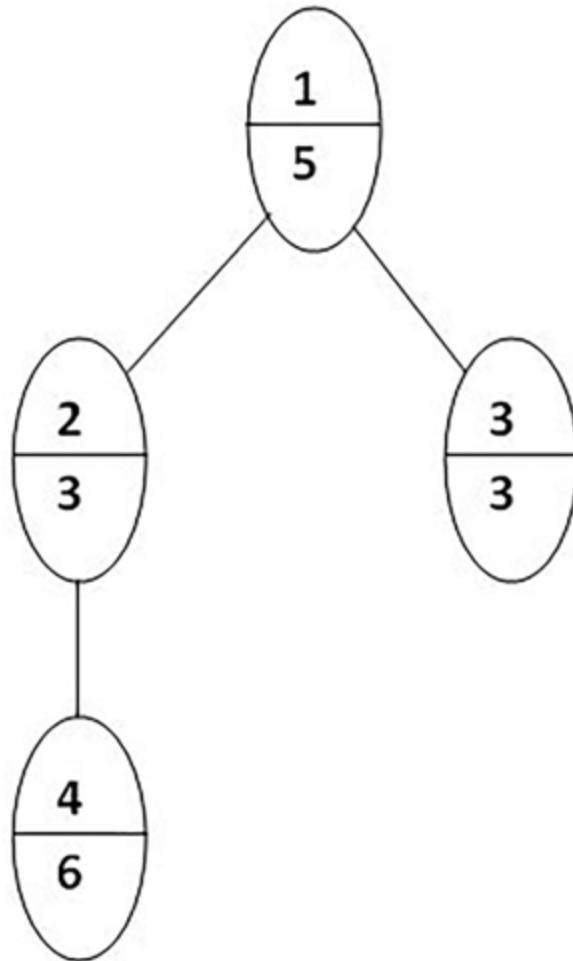
```
[[1, 5, [2, 3]], [2, 3, [4]], [3, 3, []], [4, 6, []]]
```

ID to be searched: 1

**Output:** 17

**Explanation:**

The employee tree is shown in [\*Figure 5.15\*](#). Each node shows its ID and sales values:



*Figure 5.15: Example tree*

Employee 1 has employees 2 and 3 under him. Employee 2 has employee 4 under him. The sales of employee 2 will be the sum of his own sales, and the sales of his subordinate 4, that is,  $3+6=9$ . The sales of employee 1 will be the sum of his own sales, and the sales of employees 2 and 3, that is,  $5+9+3=17$ .

## Solution format

Your solution should be in the following format:

```

1. class Solution:
2.     def getSales(self, employees: List['Employee'], id: int)
-> int:

```

## Strategy

The solution format requires the list of the objects of the class Employee. Therefore, we will have to do some work in the driver code to convert the list to the required form. It is achieved with the following lines:

```
emps,id=[[1, 5, [2, 3]], [2, 3, [4]], [3, 3, []], [4,6,[]]],1  
emp_list = []  
for el in emps:  
    emp_list.append(Employee(el[0],el[1],el[2]))
```

Now, let's come to the problem. We will use the recursive DFS strategy. Each Employee object contains a field containing the list of IDs of the subordinates. We need a quick way to locate the object pointer from the ID. So, first of all, we will build a dictionary, named emap which contains pairs (ID: Object of ID). We will initialize the total\_sales with the employee's own sales. Then, we will recursively call itself while passing the subordinates' IDs as arguments. The values returned from these calls will be added to the total\_sales.

## Python code

The following code implements this strategy:

```
1. from typing import List  
2. # Definition for Employee.  
3. class Employee:  
4.     def __init__(self, id: int, sales: int, subordinates:  
List[int]):  
5.         self.id = id  
6.         self.sales = sales  
7.         self.subordinates = subordinates  
8. class Solution:  
9.     def getSales(self, employees: List['Employee'], id: int)  
-> int:  
10.        emap = {e.id: e for e in employees}  
11.        employee = emap[id]
```

```

12.         total_sales = employee.sales
13.         for id in employee.subordinates:
14.             total_sales += self.getSales(employees, id)
15.         return total_sales
16. #Driver code
17. sol = Solution()
18. emps,id=[[1, 5, [2, 3]], [2, 3, [4]], [3, 3, []],
19.           [4,6,[]]],1
20. emp_list = []
21. for el in emps:
22.     emp_list.append(Employee(el[0],el[1],el[2]))
23. print(sol.getSales(emp_list,id))

```

**Output:** 17

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If  $n$  is the total number of employees, then each employee is visited once during the DFS query. So, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the stack requirements of recursion, the space complexity is  $O(n)$ .

## Conclusion

We had covered the basics of the graph theory in the last chapter. We took it forward with the implementation of the basic DFS technique in Python. Then we covered 12 questions on the DFS applications covering various situations. While solving these questions, we learned three important things. Firstly, how to convert a word description into a graph based-model, secondly how to apply the basic DFS algorithm to the given situation, and thirdly, how to utilize various facilities of Python to come up with an efficient solution. Consider the first question about finding the itinerary from tickets. The main skill in solving it was formulation. In the following

questions, we learned the standard graph theory terminology like symmetric trees, balanced trees, and binary search trees. We also learned how to solve problems based on these. We also covered two problems about the islands in water. You need a two-dimensional search methodology for solving these problems. In future, if you come across an unknown problem, you will be able to apply the techniques presented here. In the next chapter, we will study the breadth-first search, or the BFS.

## **Points to Remember**

- In a DFS traversal, we use either a recursion or a stack to store the nodes to be searched.
- The Defaultdict is a convenient class to use as a dictionary.

## **MCQs**

1. For getting the nodes of a BST in a sorted order, the traversal should be?
  - A. Pre-order Traversal
  - B. Post-order Traversal
  - C. Level-order Traversal
  - D. In-order Traversal
2. The time complexity of the DFS in a graph is:
  - A.  $O(V * E)$
  - B.  $O(V)$
  - C.  $O(E)$
  - D.  $O(V + E)$
3. The data structure used to store nodes to be searched in the depth-first search is?
  - A. Linked List
  - B. Queue
  - C. Stack

D. Tree

4. In the depth-first search, how many times is a node visited?
  - A. Once
  - B. Equivalent to the number of in-degree of the node
  - C. Twice
  - D. Thrice
5. For finding the shortest path between two nodes, which method is better?
  - A. DFS
  - B. BFS
  - C. Both are same
6. An undirected random graph has eight vertices. The probability that there is an edge between a pair of vertices is 1/2. What is the expected number of unordered cycles of length 3?
  - A. 7
  - B. 1
  - C. 1/8
  - D. 8
7. The height of a tree is the number of edges from the root to the farthest leaf. The maximum number of nodes in a binary tree of height  $h$  is:
  - A.  $2^h$
  - B.  $2^h - 1$
  - C.  $2^{h-1} + 1$
  - D.  $2^{h+1} - 1$
8. How many binary trees can be formed with 3 unlabeled nodes?
  - A. 3
  - B. 5

C. 30

D. 8

9. What is common in the three different types of traversals (in-order, pre-order, and post-order)?

- A. The root is visited before the right subtree
- B. The left subtree is always visited before the right subtree
- C. The root is visited after the left subtree
- D. All of the above

10. Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph?

- A. Depth-first search
- B. Breadth-first search
- C. Prim's minimum spanning tree algorithm
- D. Kruskal' minimum spanning tree algorithm

## Answers to the MCQs

**Q1:** D

**Q2:** D

**Q3:** C

**Q4:** A

**Q5:** B

**Q6:** A

**Explanation:** If we choose 3 vertices (a,b,c) out of 8 to form a cycle, there are  ${}^8C_3 = 56$  combinations. The probability of edges ab, bc, and ac being present is  $\frac{1}{2}$  each. The probability of all the three edges being present is  $(1/2)^3 = 1/8$ . Thus, the expected number length of the 3 cycles is  $56/8 = 7$ .

**Q7:** D (See Section 5.1, DFS traversal of trees)

**Q8:** B (See Section 5.1, DFS traversal of trees)

**Q9:** B

**Q10:** A

## **Questions**

1. What are the properties of a binary search tree?
2. What is a balanced tree?
3. What is a symmetric tree?
4. Why the union-find method cannot be used to detect cycles in a directed graph?
5. Explain the use of the lambda function to calculate the sorting key.

## **Key Terms**

DFS, binary search tree, union find method, directed graph, cycles.

# CHAPTER 6

## Breadth First Search

As discussed in [Chapter 4](#), there are two methods for traversing a graph, namely Depth First Search (DFS) and Breadth First Search (BFS). The BFS algorithm traverses a graph in a breadth-wise manner. So, it enters all the neighbors of a vertex in a queue. Initially, the starting vertex is entered in the queue. For trees, the root is the starting point. In the case of a graph, we need to select some arbitrary vertex as the starting node. When a dead end occurs, we get the next vertex from the queue. The DFS algorithm is similar but it stores the neighbors in a stack instead of a queue. This results in a depth-wise search.

If adjacency list is used to represent the graph, the time complexity of both algorithms is  $O(V+E)$  where  $V$  is the number of vertexes and  $E$  is the number of edges. If adjacency matrix is used to represent the graph, the time complexity of both algorithms is  $O(V^2)$ . But actual suitability depends on the type of problem. If we are searching for a target vertex which is near the root, BFS is quicker. For searching a decision tree, DFS is quicker.

### Structure

We will start with basic BFS manipulation techniques by creating a graph class in having functions of initialization, adding edges and printing the nodes in BFS order. Then we will take five questions based on BFS modeling. We will cover the topics in the following order:

- Basic BFS graph class
- Question 48: How will you print a tree level wise?
- Question 49: How will you build a word ladder between two words by changing one letter at a time?
- Question 50: How will you find a course sequence according to pre-requisites?

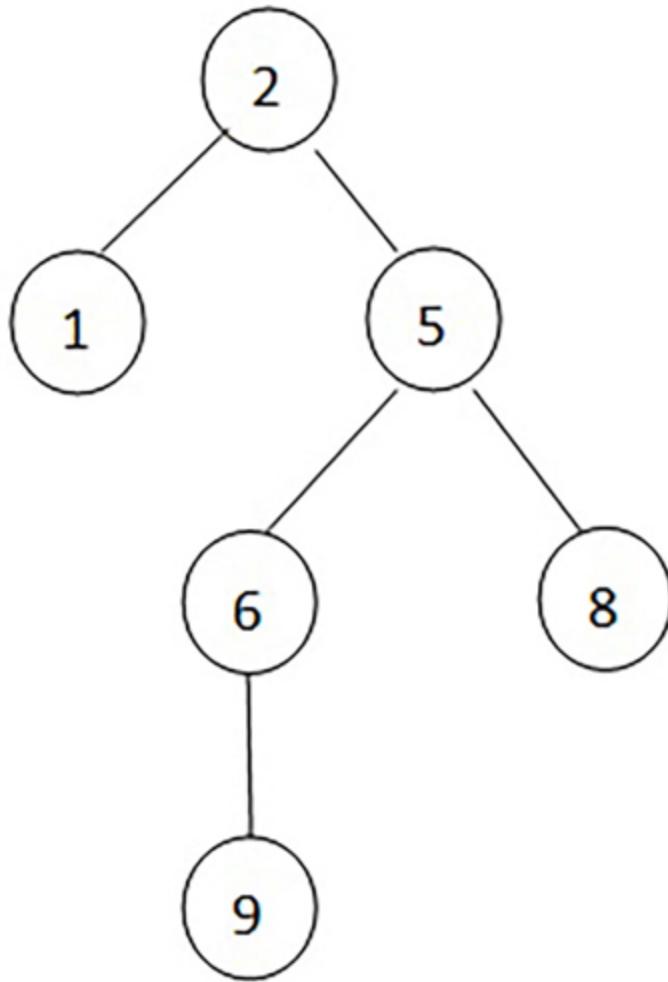
- Question 51: How will you test if two nodes of a tree are cousins?
- Question 52: How will you find the shortest bridge across two islands?

## **Objectives**

After studying this unit, you will be familiar with the BFS technique. You will be able to recognize problems which are suitable for BFS modelling. You will be confident of solving problems on BFS.

### **6.1 Basic BFS graph class**

We will start with basic BFS manipulation techniques by creating a BFS graph class having functions of initialization, adding edges and printing the nodes in BFS order. Let us use the example graph of [Figure 6.1](#) for illustrating this class:



*Figure 6.1: Example graph*

The initialization function of the class creates an empty dictionary named `graph` to store the graph in adjacency list form.

The `addEdge(u, v)` function appends node `v` in the adjacency list of `u`, that is, `graph[u]`.

The `bfs` function prints the node values in BFS order. There are two approaches for implementing `bfs`, namely **iterative** and **recursive**. Both approaches are presented here.

In the iterative approach, we will create a queue named `q` to store the nodes to be visited. We will also create a set named `visited` to keep track of visited nodes. Initially we will push the root `s` into `q`. Then we will start a while loop which runs as long as there are elements in `q`. In the loop, we

will pop an element from q into u and print it. Then we will scan the adjacency list of u and if a node in the list has not been visited, then we will append it to the queue.

## Python code

Following code implements the Graph class described earlier:

```
1. #BFS iterative approach
2. from collections import defaultdict
3. class Graph:
4.     def __init__(self):
5.         self.graph = defaultdict(list)
6.
7.     def addedge(self, u, v):
8.         self.graph[u].append(v)
9.
10.    def bfs(self, s):
11.        visited = set()
12.        queue = []
13.        queue.append(s)
14.        visited.add(s)
15.        while queue:
16.            u = queue.pop(0)
17.            print(u, end=" ")
18.            for v in self.graph[u]:
19.                if v not in visited:
20.                    queue.append(v)
21.                    visited.add(v)
22.
23. g = Graph()
24. g.addedge(2, 1)
25. g.addedge(2, 5)
26. g.addedge(5, 6)
27. g.addedge(5, 8)
```

```
28. g.addedge(6, 9)
29.
30. g.bfs(2)
```

### Output:

```
2 1 5 6 8 9
```

### Recursive approach

The `bfs` function can also be implemented in a recursive manner. For this, we will define a recursive helper function `recursiveBFS`. It does not take any arguments other than `self`. It will scan the tree structure in `self` and print the node values. The `bfs` function will initialize `q` to root node, `visited` set to empty and call `recursiveBFS`. The exit condition for recursion will be the queue being empty. The `recursiveBFS` function will pop elements from the queue, print them, append its neighbors to queue and call itself.

### Python code

Following code implements the recursive approach. Driver code from the iterative approach can be reused:

```
1. def bfs(self, node):
2.     self.visited = set()
3.     self.q = deque()
4.     self.q.append(node)
5.     self.recursiveBFS()
6.
7.     def recursiveBFS(self):
8.         if not self.q:
9.             return
10.        # pop front node from queue and print it
11.        v = self.q.popleft()
12.        print(v, end=' ')
13.        # do for every edge (v -> u)
14.        for u in self.graph[v]:
```

```
15.         if u not in self.visited:
16.             # mark it visited and push it into queue
17.             self.visited.add(u)
18.             self.q.append(u)
19.         self.recursiveBFS()
```

### **Output:**

2 1 5 6 8 9

## **6.2 Question 48 – How will you print a tree level-wise?**

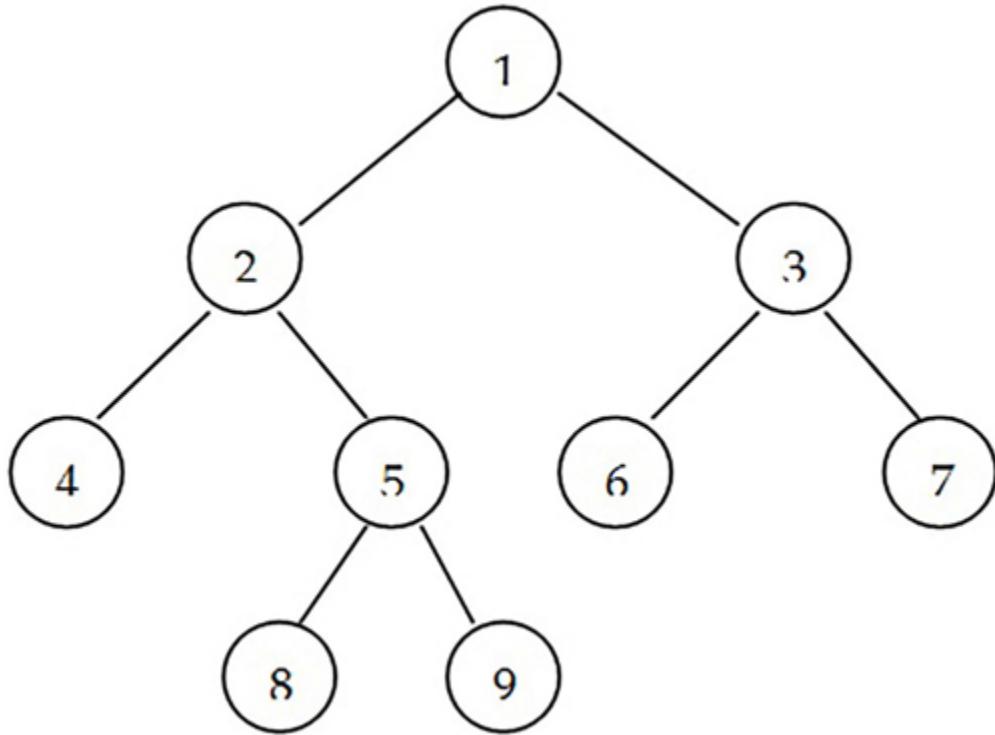
We had presented a similar problem using recursive technique in section 4.1 during introduction to trees and graphs. Now we present a queue based iterative technique. In that problem, node values were printed. In this one, node values will be returned as a list of level-wise lists.

### **Problem statement**

You are given the root of a tree. Return a list of lists such that each list contains the node values in one level. Let us illustrate with the help of an example.

#### **Example:**

Consider the tree shown in [\*Figure 6.2\*](#):



*Figure 6.2: Example graph*

The function should return following list of lists:

```
[[1], [2, 3], [4, 5, 6, 7], [None, None, 8, 9, None, None, None, None]]
```

## Solution format

Your solution should be in following format:

```
1. class Solution:
2.     def levelOrder(self, root: TreeNode) -> List[List[int]]:
```

## Strategy

As per the classic BFS approach, we will initialize a queue to store the nodes to be visited. Initially, we will push root into the queue. We will create an empty list named `List` to which we will append level-wise lists.

Now we will run a `while` loop which will continue as long as there is something in the queue. We will create an empty list named `LvlList` to

which we will append node values of this level. When a level is finished, we will append `LvlList` to `List`. Now the crucial point is, “How will you know that one level has ended in the stream coming out of the queue?” We will make use of an important property of the Python for-loop.

**The number of iterations to be done by a for-loop is decided right in the beginning of the loop. During the course of execution of the loop, even if the variables defining the range change, the number of iterations remains same.**

Inside the while loop, we will run a nested for-loop where index `i` goes from 0 to length of the queue -1. We will push more items into the queue during the execution of the loop. These items belong to the next level. This increases the queue length but it does not affect the original number of repetitions of the loop. We will pop nodes from the queue one by one. First we append the node value to the `LvlList` and then push its left and right children into the queue if they exist. After the inner loop is finished, we will append `LvlList` to `List`.

## Python code

The following code implements the strategy described. The driver code from section 4.1 has been reused:

```
1. from typing import List
2. class TreeNode:
3.     def __init__(self, v):
4.         self.val = v
5.         self.left = None
6.         self.right = None
7.
8. class Solution:
9.     def levelOrder(self, root: TreeNode) -> List[List[int]]:
10.         if root is None: return root
11.         q = []
12.         List = []
13.         q.append(root)
```

```

14.         while len(q):
15.             LvlList = []
16.             for i in range(len(q)):
17.                 node = q.pop(0)
18.                 LvlList.append(node.val)
19.                 if node.left != None:
20.                     q.append(node.left)
21.                 if node.right != None:
22.                     q.append(node.right)
23.             List.append(LvlList)
24.         return List
25. def makeTree(arr:List[int])-> TreeNode:
26.     return NodeInsert(arr,None,0)
27. def NodeInsert(arr:List[int],root:TreeNode,i:int)->
TreeNode:
28.     n=len(arr)
29.     if i<n:
30.         root = TreeNode(arr[i])
31.         # insert left child
32.         root.left = NodeInsert(arr, root.left, 2 * i + 1)
33.         # insert right child
34.         root.right = NodeInsert(arr, root.right, 2 * i + 2)
35.     return root
36.
37. root =
makeTree([1,2,3,4,5,6,7,None,None,8,9,None,None,None,None])
38. sol=Solution()
39. L = sol.levelOrder(root)
40. print(L)

```

## **Output:**

[[1], [2, 3], [4, 5, 6, 7], [None, None, 8, 9, None, None, None, None]]

## Complexity

The complexity analysis is as follows:

**Time complexity:** If the number of nodes is  $n$ , the while-loop runs  $n$  times. Thus the time complexity is  $O(n)$ .

**Space complexity:** Length of the queue is equal to the maximum width of the tree  $w$ . The space complexity is therefore  $O(w)$ .

## 6.3 Question 49 – How will you build a word ladder between two words by changing one letter at a time?

This example teaches you how to formulate a graph theory based model from word description. Since we are interested in the shortest path, BFS is the correct approach. We will also do a comparative study of various classes in Python that can implement a queue.

### Problem statement

You are given two strings named `beginWord` and `endWord`. You are also given a list of valid words named `wordList`. You have to build a word ladder from `beginWord` to `endWord` using words from `wordList` in such a manner that two consecutive words in the ladder differ by only one character (for example, lot and dot). You have to find the length of the shortest transformation sequence. If the sequence does not exist, then you should return 0.

In addition, you may assume that all words in the word list are of the same length, they are in lower case and there are no duplicates.

Let us illustrate with the help of an example.

#### **Example:**

Let `beginWord` be “hit”, `endWord` be “cog” and `wordList` be[“hot”, “dot”, “dog”, “lot”, “log”, “cog”]

We can build a ladder, hit->hot->lot->log->cog.

This ladder has five elements, so we should return 5.

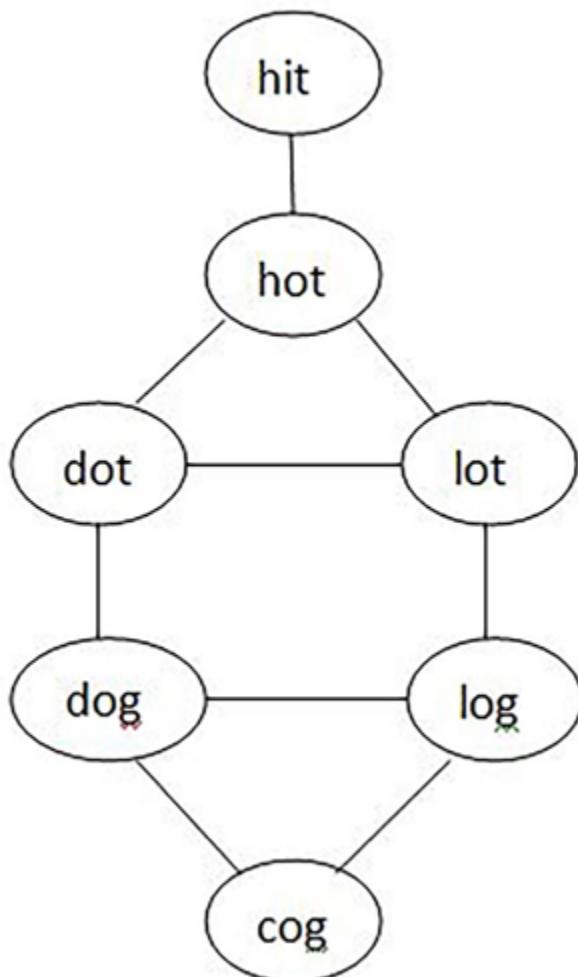
### Solution format

Your solution should be in the following format:

```
1. class Solution(object):  
2.     def  
ladderLength(self, beginWord:str, endWord:str, wordList:List[str])  
-> int:
```

## Strategy

We will represent words as nodes of a graph. We will connect two nodes if they differ by one letter only. The graph corresponding to the example is shown in [Figure 6.3](#):



*Figure 6.3: Example graph*

Our problem now reduces to finding the shortest path from hit to cog. Notice that the shortest path may not be unique, but we are interested only in its length. One way of solving this problem would be to represent the graph in adjacency list form and then apply the standard shortest distance algorithm. But it is more efficient to find the adjacent neighbors directly.

For efficiently finding neighbors, we will build a dictionary named `wildcard_dict`. Each node in the example can be represented by three wild card names, for example, “hot” can be represented as “\*ot”, “h\*t” and “ho\*”. We will make three entries into the dictionary mapping the wild card notation to the original word. The entries will be:

```
*ot: hot
h*t: hot
ho*: hot
```

The expression `word[:i] + “*” + word[i+1:]` replaces  $i^{\text{th}}$  letter in the word with \*.

The contents of the wild card dictionary are given in [Table 6.1](#):

Key	Value
*og	[‘dog’, ‘log’, ‘cog’]
*ot	[‘hot’, ‘dot’, ‘lot’]
c*g	[‘cog’]
co*	[‘cog’]
d*g	[‘dog’]
d*t	[‘dot’]
do*	[‘dot’, ‘dog’]
h*t	[‘hot’]
ho*	[‘hot’]
l*g	[‘log’]
l*t	[‘lot’]
lo*	[‘lot’, ‘log’]

**Table 6.1:** Contents of wild card dictionary

After building the dictionary, we will create a queue to store the future nodes to be visited as per the standard BFS procedure.

Let us now take a little detour to discuss two methods in Python that can implement a queue. We have been using the List class in many problems. It can act as a queue. The append function enters an item into the queue and pop(0) function retrieves the item that was entered first. But the time complexity of the pop(0) operation is high. It is  $O(n)$ . The List class is suitable for implementing a stack because the time complexity of both append (that is, push) and pop operations is  $O(1)$ . For efficient implementation of the queue, the Python module “collections” offers a class “deque”(meaning double ended queue). It is a generalization of queue and stack classes. We can push or pop items at both ends of a deque with complexity  $O(1)$ . Internally, it uses a doubly linked list for implementation. For queue operation, we can use append and popleft functions of the deque class.

As we want to keep track of the path length, we will push a tuple (node, level) into the queue. The first entry will be (`beginWord`, 1), that is, in our case, (“hit”, 1).

We will also create a set named `visited` to store a list of visited nodes.

Then we will run a while loop as long as the queue is not empty. In the loop, we will pop the queue head into a variable `popped_word` and its level in variable `level`. The first word to be popped will be “hit”. From this word we will create its wild card versions named `intermediate_word`, i.e. “\*it”, “h\*t” and “hi\*”. We will retrieve the list of original words corresponding to `intermediate_word` from `wildcard_dict`. Now the keys “\*it” and “hi\*” don’t exist in [Table 6.1](#) but “h\*t” has a value “hot”. Likewise, the key “\*og” gives a list of three original words, namely [`dog`, `log`, `cog`]. Let us call it the `originals` list. We will iterate the `originals` list and each word from the list will be named `word`.

If the word matches `endword`, then we have found the destination. We will return the answer which will be obtained by adding 1 to `level`. If we don’t find a match, we will see if the word has been previously visited. If not, then we will add it to the `visited` set and also enter the tuple (`word`,

`level+1`) in the queue. When the originals list is done, we will wipe out the entry corresponding to `intermediate_word` from `wildcard_dict`.

If the queue becomes empty and we have still not hit the `endWord`, then we should return 0.

## Python code

Following code implements the strategy described above:

```
1. from typing import List
2. from collections import defaultdict, deque
3. class Solution(object):
4.     def ladderLength(self, beginWord: str, endWord: str,
wordList: List[str]) -> int:
5.         if endWord not in wordList or not endWord or not
beginWord or not wordList:
6.             return 0
7.         # Find the length of the words
8.         L = len(beginWord)
9.
10.        # Build Wildcard dictionary of the form
11.        # {*at:hat,h*t:hat,ha*:hat}
12.        wildcard_dict = defaultdict(list)
13.        for word in wordList:
14.            for i in range(L):
15.                # Key is the generic word (wild card)
16.                # Value is a list of words which have the same
wild card
17.                #representation.
18.                wildcard_dict[word[:i] + "*" +
word[i+1:]].append(word)
19.        # Initialize Queue for BFS with beginWord, level 1
20.        queue = deque([(beginWord, 1)])
21.        # Maintain a list of visited words, initially
beginWord
```

```

22.         visited = {beginWord}
23.         while queue:
24.             #for i in range(len(queue)):print(queue[i])
#For debugging
25.             popped_word, level = queue.popleft()
26.             for i in range(L):
27.                 # Intermediate words *at,h*t,ha*
28.                 intermediate_word = popped_word[:i] + "*" +
popped_word[i+1:]
29.                 #Search intermediate word in wildcard
dictionary
30.                 for word in
wildcard_dict[intermediate_word]:
31.                     if word == endWord: #Found the word?
32.                         return level + 1#Yes, return
level+1
33.                     # Otherwise, add it to the BFS Queue.
Also mark it visited
34.                     if word not in visited:
35.                         visited.add(word)
36.                         queue.append((word, level + 1))
37.                         wildcard_dict[intermediate_word] = []
38.             return 0
39. #Driver code
40. sol=Solution()
41. beginWord = "hit"
42. endWord = "cog"
43. wordList = ["hot","dot","dog","lot","log","cog"]
44. print(sol.ladderLength(beginWord,endWord,wordList))

```

## **Output:**

5

## **Complexity**

The complexity analysis is as follows:

**Time complexity:** Suppose the number of words is  $n$ , and the length of each word is  $m$ . For building the wild card dictionary, the outer loop runs  $n$  times and the inner loop runs  $m$  times. Each iteration has a string operation of complexity  $O(m)$ . Thus the overall time complexity is  $O(m^2 n)$ .

For finding the shortest path, the while loop runs  $n$  times and the inner loop runs  $m$  times. Each iteration has a string operation of complexity  $O(m)$ . Thus the overall time complexity is  $O(m^2 n)$ .

Combining the two, the time complexity of is  $O(m^2 n + m^2 n) = O(m^2 n)$ .

**Space complexity:** Length of the wild card dictionary is  $m*n$ . Each entry in the dictionary may have at the most  $m$  entries. Thus, the space complexity is  $O(m^2 n)$ .

The queue has  $n$  entries, each of length  $m$ . So the space complexity is  $O(m n)$ . Similarly, the visited set has space complexity is  $O(m n)$ . Thus, the overall space complexity is  $O(m^2 n + mn + mn) = O(m^2 n)$ .

## 6.4 Question 50 – How will you find a course sequence according to pre-requisites?

We had done a similar question, question 46 in [Chapter 5](#). The goal was to find whether we can finish all courses subject to the pre-requisite conditions. We solved it using DFS approach. Here we not only want to find the validity of the schedule; we also want the valid sequence of courses. We will use BFS technique.

### Problem statement

A university runs various courses that have to be taken in a particular sequence. Every course  $c$  has a pre-requisite  $p$ . This dependency relationship is described as  $[c, p]$ . You are given a list of the dependency relationships. Write a function to return a valid sequence of courses. A valid sequence may not be unique. You may return any one of them. If there is no valid sequence, then the returned list will be empty.

Let us illustrate this with the help of an example.

## **Example:**

**Input:** `[[1, 0], [2, 0], [3, 1], [3, 2]]`

**Output:** `[0, 1, 2, 3]` or `[0, 2, 1, 3]`

**Explanation:** 0 should be done before 1 and 2. 1 or 2 should be done before 3. These conditions are satisfied by `[0,1,2,3]` and `[0,2,1,3]`.

## **Solution format**

Your solution should be in following format:

```
1. class Solution:  
2.     def findOrder(self, numCourses:int,  
prerequisites>List[List[int]])->List[int]:
```

## **Strategy**

We will model the problem with pre-requisite conditions as directed edges and courses as nodes. Courses that do not have any pre-requisites are the starter courses. They can be identified as nodes having 0 in-degree. We will create a dictionary named `indegree` to store the in-degrees of all nodes. We will scan the prerequisites list and fill the `indegree` dictionary. The starter courses will not be included in `indegree`. Then we will put all the starter courses in a queue named `q`. We will create an empty list named `course_sequence` to store the answer.

Then we will run a `while` loop as long as the queue is not empty. In this loop, we will pop an element from the `q` into a variable named `vertex`. We will append the `vertex` to `course_sequence` and remove it from the graph along with its outgoing edges, that is, reduce the in-degree of all its neighbors by 1. By doing so, some neighbor's in-degree may reduce to 0. We will add it to `q`.

When `q` becomes empty, `course_sequence` contains the valid sequence. But it is possible that some unconnected nodes may be left out. So we should check that the length of `course_sequence` is equal to the number of courses. If we are unable to find 0 in-degree nodes, then `course_sequence` will remain empty.

The course\_sequence so ordered is called topological ordering. Topological ordering of a directed graph is a list of its vertexes such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

## Python code

Following code implements the strategy described above:

```
1. from collections import defaultdict, deque
2. from typing import List
3. class Solution:
4.     def findOrder(self, numCourses: int,
5.                   prerequisites: List[List[int]]) ->
6.                 List[int]:
7.         # Build adjacency list
8.         adj_list = defaultdict(list)
9.         indegree = {}
10.        for dest, src in prerequisites:
11.            adj_list[src].append(dest)
12.            # Increment dest node's in-degree
13.            # If node does not exist, default value is 0
14.            indegree[dest] = indegree.get(dest, 0) + 1
15.        # Queue of nodes having 0 in-degree
16.        q = deque([k for k in range(numCourses) if k not in
17.                  indegree])
18.
19.        course_sequence = []
20.
21.        # Until there are nodes in the Q
22.        while q:
23.
24.            # Pop one node with 0 in-degree
25.            vertex = q.popleft()
26.            course_sequence.append(vertex)
```

```

25.
26.        # Reduce in-degree for all the neighbors
27.        if vertex in adj_list:
28.            for neighbor in adj_list[vertex]:
29.                indegree[neighbor] -= 1
30.
31.        # Add neighbor to Q if in-degree
becomes 0
32.        if indegree[neighbor] == 0:
33.            q.append(neighbor)
34.
35.    return course_sequence if\
36.        len(course_sequence) == numCourses else []
37. #Driver code
38. sol=Solution()
39. pre=[[0,1],[1,3],[1,4],[1,5],[2,6],[2,7]]
40. print(sol.findOrder(8,pre))

```

## **Output:**

[3, 4, 5, 6, 7, 1, 2, 0]

## **Complexity**

The complexity analysis is as follows:

**Time complexity:** Suppose the number of courses is  $n$ , and the number of pre-requisite conditions is  $p$ . For building the adjacency list, we run the loop  $p$  times. For finding the course sequences, the outer while loop runs  $n$  times. The inner loop runs as many times as there are adjacencies, which could be at worst  $n$ .

Combining the two, the time complexity of is  $O(p + n^2) = O(n^2)$ .

**Space complexity:** Length of the adjacency list is at worst  $p$  with each entry having at worst  $n$  elements. However, the total is bound by  $n$ , its space complexity is  $O(n)$ . Length of the queue can be at worst  $n$ . Combining the two, the time complexity of is  $O(n)$ .

## 6.5 Question 51 – How will you test if two nodes of a tree are cousins?

This is similar to question 33 of [Chapter 4](#), “How will you find lowest common ancestor in a binary tree”. In that question, the goal was to find a lowest level common ancestor of two given nodes. Here we want to find if the two nodes are cousins, that is, same level but different parent.

### Problem statement

Consider a multi-level tree. The root is at depth 0. The children of root are at depth 1. In general, the children of depth  $n$  nodes are at depth  $n+1$ . Let us call the children of a node as brothers and the children of brothers as cousins. Two nodes will be called cousins if they are at same depth but their parents are different. Write a function which will determine if the two given nodes are cousins. Nodes are identified by their value, which is an integer.

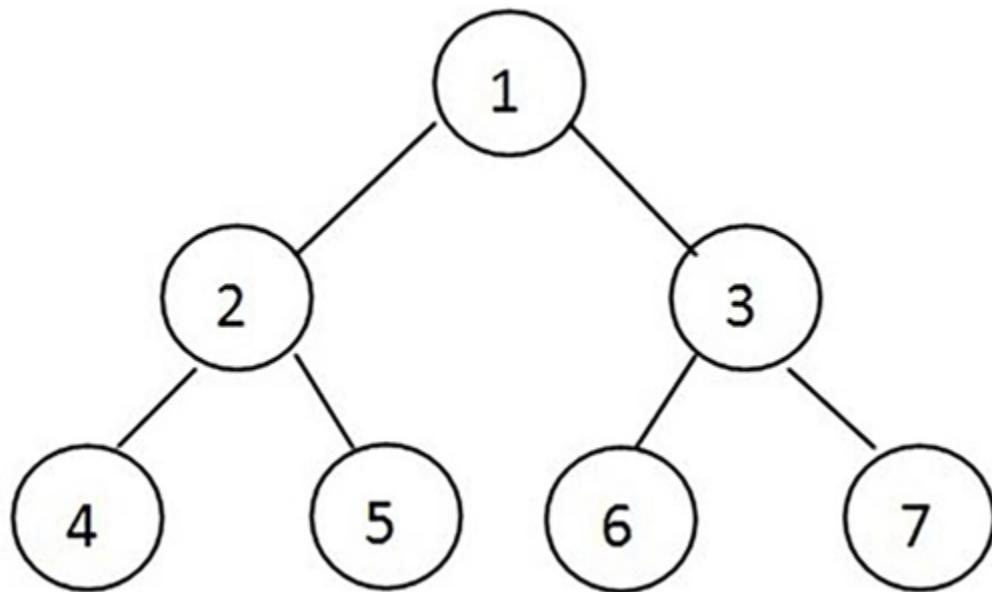
Let us illustrate with the help of an example.

#### **Example:**

**Input:** Tree = [1,2,3,4,5,6,7] nodes are 4 and 7

**Output:** True

**Explanation:** The example tree is shown in [Figure 6.4](#):



*Figure 6.4: Example graph*

4 and 7 are both at depth 2 and their parents are 2 and 3 respectively. So they are cousins.

For the same graph, output for [4,5] will be false and out for [2,7] will be false.

## Solution format

Your solution should be in following format:

```
1. class Solution:  
2.     def isCousins(self, root: TreeNode, x: int, y: int) ->  
    bool:
```

## Strategy

In question 33 of [Chapter 4](#) we built a dictionary to store the parents of nodes. Here we are interested in depths as well, so we will store the parent and depth of the node in the dictionary named `par_depth`. We will use BFS strategy as it is convenient for depth wise traversal. We will create a queue named `q` and initialize it with the tuple(`root,0`).

Then we will run a `while` loop as long as the queue is not empty. In the loop, we will pop the front of the queue into the variables `node` and `depth`. If the node has a left child, we will enter the tuple `(node.val, depth+1)` at index `node.left.val` in the dictionary `par_depth` and append the tuple `(node.left, depth+1)` to `q`. Similarly if the node has a right child, we will enter the tuple `(node.val, depth+1)` at index `node.right.val` in the dictionary `par_depth` and append the tuple `(node.right, depth+1)` to `q`. In our example, `root` has two children namely 2 and 3. After the above step, the contents of the dictionary are:

```
par_depth = {2:(1,1), 3:(1,1)}
```

The contents of the `q` are:

```
q = node 2,(1,1), node 3,(1,1)
```

In the next iteration, we will pop node 2 and repeat. We will break from the loop if x and y are found. At the end of the loop, if any one of x and y remains untraced, then we will return false. After coming out of loop, we check if the depths of x and y are same and their parents are different. If so, then we will return true, else false.

## Python code

Following code implements the strategy described above. The driver code from previous problems has been reused:

```
1. from typing import List
2. from collections import deque
3. class TreeNode:
4.     def __init__(self, v):
5.         self.val = v
6.         self.left = None
7.         self.right = None
8.     def printTree(root:TreeNode)->None:
9.         LevelList = [root]
10.        printLevel(LevelList)
11.
12.    def printLevel(LevelList:List[TreeNode])-> List[TreeNode]:
13.        LevelStr = ""
14.        outList = []
15.        ListEmpty = True
16.        for node in LevelList:
17.            if node is None:
18.                LevelStr += "None "
19.                outList.append(None)
20.                outList.append(None)
21.            else:
22.                LevelStr += (str(node.val) + " ")
23.                outList.append(node.left)
24.                outList.append(node.right)
```

```

25.         ListEmpty = False
26.     if not ListEmpty:
27.         print(LevelStr)
28.         printLevel(outList)
29. class Solution:
30.     def isCousins(self, root: TreeNode, x: int, y: int) ->
bool:
31.         q = deque()
32.         q.append((root,0))
33.         par_depth={}
34.         while q:
35.             node,depth = q.popleft()
36.             if node.left:
37.                 par_depth[node.left.val]=(node.val,depth+1)
38.                 q.append((node.left,depth+1))
39.             if node.right:
40.                 par_depth[node.right.val]=
(node.val,depth+1)
41.                 q.append((node.right,depth+1))
42.             if (x in par_depth) and (y in par_depth):break
43.             if x not in par_depth or y not in par_depth: return
False
44.             x_parent,x_depth = par_depth[x]
45.             y_parent,y_depth = par_depth[y]
46.             return x_depth == y_depth and x_parent != y_parent
47. #Driver code
48. root =TreeNode(1)
49. root.left =TreeNode(2)
50. root.right =TreeNode(3)
51. root.left.left = TreeNode(4)
52. root.left.right =TreeNode(5)
53. root.right.left =TreeNode(6)
54. root.right.right =TreeNode(7)
55. printTree(root)

```

```
56.  
57. sol = Solution()  
58. print(sol.isCousins(root,5,6))
```

### **Output:**

```
1  
2 3  
4 5 6 7  
True
```

### **Complexity**

The complexity analysis is as follows:

**Time complexity:** Suppose the number of nodes is  $n$ . The outer while loop runs  $n$  times. So, the time complexity is  $O(n)$ .

**Space complexity:** Length of the queue  $n$ , so its space complexity is  $O(n)$ .

## **6.6 Question 52 – How will you find the shortest bridge across two islands?**

This is another question in the island category. Here the recommended approach is a combination of DFS and BFS.

### **Problem statement**

You are given 2D array A, which represents a lake. Water is represented by 0 and land is represented by 1. The lake has two islands. (An island is a connected group of 1s surrounded by 0's. Connection is done in horizontal and vertical direction.)

Now, we want to build a bridge to connect the two islands. This can be done by flipping some 0's to 1's. Find the smallest number of 0s that must be flipped. (It is guaranteed that the answer is at least 1.) Let us illustrate with the help of an example.

### **Example:**

#### **Input:**

```
A=    [[1,1,0,0,0]
      ,[1,1,0,0,0]
      ,[0,0,0,0,0]
      ,[0,0,0,1,1]
      ,[0,0,0,1,1]]
```

**Output:** 3

**Explanation:** There is one island in the top left corner and another in the bottom right corner. We can bridge them by flipping at least three 0 elements.

## Solution format

Your solution should be in the following format:

1. class Solution:
2. def shortestBridge(self, A: List[List[int]]) -> int:

## Strategy

The solution will consist of two steps. In the first step, we will detect one of the islands using DFS method and replace 1 in the island cells with 2. In the second step, using BFS, we will grow this island on all sides till it touches the second island. The number of growth steps will give the minimum length of the bridge.

Let us discuss the first step in detail. We will run a nested loop in *i* and *j* to scan the matrix *A* to detect a 1. When the first 1 is found, we will call the function *dfs* for further work and break from the loops. But how do we break from both loops? We will use a variable *found* for this purpose. When 1 is detected in the inner loop, it sets *found*. It is detected by the outer loop and that also exits. On exit, the *dfs* function would have converted the whole island to 2.

The *dfs* function receives *A*, *i*, *j*, *visited* and *que* as arguments because these are local variables of the function *shortestBridge* and we want them to be accessed in *dfs*. The job of the function is to recursively search the neighbors of *i*, *j* and when a 1 is found, add it to the queue and set the

value to 2. We will use a matrix named `visited` to prevent visiting a node again. Refer to question 44 where we called the recursive function four times to search the neighbors in four directions. The code is repeated here:

```
#Recursive nested function to wipe out neighbours of (row,col)
def set_zero(row, col, grid):
    if (0 <= row <= nrow-1) and (0 <= col <= ncol-1):
        if grid[row][col] == '1':
            grid[row][col] = '0'
            set_zero(row+1, col, grid)
            set_zero(row-1, col, grid)
            set_zero(row, col+1, grid)
            set_zero(row, col-1, grid)
```

In this solution, we will use another method in which the call to the recursive function is only at one place but it is put inside a loop that executes four times. Both methods need same amount of typing and both methods are popular. Probably the second method will be better if we allow diagonal elements also to be neighbors. The code for present solution is:

```
dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
if A[i][j] == 1:
    que.append((i, j))
    A[i][j] = 2
    for d in dirs:
        x, y = i + d[0], j + d[1]
        if 0 <= x < M and 0 <= y < N:
            self.dfs(A, x, y, visited, que)
```

The array `dirs` contains four tuples that specify coordinate increments required for moving towards right, left, bottom and top directions.

At the end of the first step, the above loop has captured the island in the top left corner. The matrix **A** looks like this:

```
A=[[2,2,0,0,0],
 , [2,2,0,0,0]]
```

```
, [0, 0, 0, 0, 0]
, [0, 0, 0, 1, 1]
, [0, 0, 0, 1, 1]]
```

The array `visited` looks like this:

```
visited = [[1, 1, 1, 0, 0]
, [1, 1, 1, 0, 0]
, [1, 1, 0, 0, 0]
, [0, 0, 0, 0, 0]
, [0, 0, 0, 0, 0]]
```

The que looks like this:

```
que =([(0, 0), (1, 0), (1, 1), (0, 1)])
```

Now let us discuss the second step. The coordinates of elements that are marked as 2 in the matrix A have been entered in queue que. The corresponding positions in visited matrix have been set to 1. Now we will grow this island by converting surrounding 0's into 2's.

For this, we will run a while loop as long as q has elements. We need to keep track of the number of growing steps. Therefore, we will run an inner loop as many times as the length of the queue. The step count will be incremented at the end of this loop. In this loop, we will first pop the head of the queue into variables i and j. We will set `visited[i][j]` to 1. If `A[i][j]` is already found to be 1, then we have hit the second island and we will return. Otherwise we will proceed to the neighbors and convert 0's to 2's and append the coordinates to the queue.

At the end of step 1, matrix **A** looks like this:

```
A= [[2, 2, 2, 0, 0]
, [2, 2, 2, 0, 0]
, [2, 2, 0, 0, 0]
, [0, 0, 0, 1, 1]
, [0, 0, 0, 1, 1]]
```

At the end of step 2, matrix A looks like this:

```
A= [[2,2,2,2,0]
 , [2,2,2,2,0]
 , [2,2,2,0,0]
 , [2,2,0,1,1]
 , [0,0,0,1,1]]
```

At the end of step 3, matrix A looks like this:

```
A= [[2,2,2,2,2]
 , [2,2,2,2,2]
 , [2,2,2,2,0]
 , [2,2,2,1,1]
 , [0,0,0,1,1]]
```

In the next iteration, we will find A[3][3] to be 1 and we will return 3.

## Python code

Following code implements the strategy described:

```
1. from typing import List
2. from collections import deque
3. class Solution:
4.     def shortestBridge(self, A: List[List[int]]) -> int:
5.         M, N = len(A), len(A[0])#M= num rows, N=num cols
6.         dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
7.         visited = [[0] * N for _ in range(M)]#M X N matrix
8.         found = False
9.         que = deque()
10.        #Find first island
11.        for i in range(M):
12.            if found: break
13.            for j in range(N):
14.                if A[i][j] == 1:
15.                    self.dfs(A, i, j, visited, que)
16.                    found = True
```

```

17.                     break
18.         #Grow this island till it touches the second
19.         step = 0      #Initialize growth steps to 0
20.         while que:
21.             size = len(que)
22.             for _ in range(size):
23.                 i, j = que.popleft()
24.                 for d in dirs:
25.                     x, y = i + d[0], j + d[1]
26.                     if 0 <= x < M and 0 <= y < N:
27.                         visited[x][y] = 1
28.                         if A[x][y] == 1:
29.                             return step
30.                         elif A[x][y] == 0:
31.                             A[x][y] = 2
32.                             que.append((x, y))
33.                         else:
34.                             continue
35.             step += 1
36.         return -1    #If you don't find another island,
return -1
37.     #Recursive function to collect land pieces around i,j
38.     def dfs(self, A, i, j, visited, que):
39.         if visited[i][j]: return
40.         visited[i][j] = 1
41.         M, N = len(A), len(A[0])
42.         dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
43.         if A[i][j] == 1:
44.             que.append((i, j))
45.             A[i][j] = 2
46.             for d in dirs:
47.                 x, y = i + d[0], j + d[1]
48.                 if 0 <= x < M and 0 <= y < N:
49.                     self.dfs(A, x, y, visited, que)

```

```
50. #Driver code  
51. sol=Solution()  
52. A = [[1,1,0,0,0],[1,1,0,0,0],[0,0,0,0,0],[0,0,0,1,1],  
[0,0,0,1,1]]  
53. print(sol.shortestBridge(A))
```

Output: 3

## Complexity

The complexity analysis is as follows.

**Time complexity:** If the dimensions of A are M and N, the DFS search is executed MN times and the BFS loop is also executed MN times. Therefore, the time complexity is  $O(n)$ .

**Space complexity:** As we are maintaining a queue of worst case length MN and a visited matrix of MN elements, the space complexity is  $O(MN)$ .

## Conclusion

We started with implementation of basic BFS class in Python. Then we covered 5 questions on BFS applications. We also introduced the deque class in Python for easy implementation of queues.

Question 48 on level wise printing was a straight forward application of the BFS method using a queue. Question 49 on finding a word ladder was very interesting. First we had to formulate the problem in terms of graph theory. Then we decided not to represent it in the standard adjacency list form. We found adjacent nodes by wild card method. We used string manipulation features of Python for this. Then we used BFS method to build the ladder. Question 50 asked you to find valid sequence of courses. The main trick was to formulate the problem in terms of graph theory. In question 51, we had to find if two nodes were cousins, i.e. a same level. Whenever we are comparing levels, we should think of BFS method. Question 52 was based on water and islands. The scenario was similar to what we studied in questions 44 and 45 of the last chapter. As we had to find shortest distance between two regions, BFS method was selected.

In the next chapter we will study backtracking.

## Points to remember

- In BFS traversal, we use a queue store the nodes to be searched.
- BFS is more efficient when the item to be searched is near the starting point.
- Collections.deque can be used for fast implementation of a queue.
- The expression word[:i] + “\*” + word[i+1:]replaces i<sup>th</sup> letter in the word with \*.

## MCQs

1. Breadth First Search is equivalent to which of these traversal in the Binary Trees?
  - A. Pre-order Traversal
  - B. Post-order Traversal
  - C. Level-order Traversal
  - D. In-order Traversal
2. The Data structure used in standard implementation of Breadth First Search is?
  - A. Stack
  - B. Queue
  - C. Linked List
  - D. Tree
3. When the Breadth First Search of a graph is unique?
  - A. When the graph is a Binary Tree
  - B. When the graph is a Linked List
  - C. When the graph is a n-ary Tree
  - D. When the graph is a Ternary Tree
4. In Depth First Search, how many times is a node visited?
  - A. Once

- B. Equivalent to number of in-degree of the node
  - C. Twice
  - D. Thrice
5. Time complexities of append and popleft operations of deque class are:
- A.  $O(n)$ ,  $O(n)$
  - B.  $O(1)$ ,  $O(n)$
  - C.  $O(1)$ ,  $O(1)$
  - D.  $O(n)$ ,  $O(1)$

## Answers to MCQs

**Q1:** C

**Q2:** B

**Q3:** B

**Q4:** A

**Q5:** C

## Questions

1. Discuss merits and demerits of breadth first search.
2. What is topological ordering?
3. Why it is preferable to use deque class to implement a queue in BFS?

## Key terms

BFS, queue, deque, level order traversal

# CHAPTER 7

## Backtracking

In this chapter, we will study a useful algorithmic technique, namely backtracking.

Backtracking incrementally builds candidates to the solutions, and abandons a candidate (*backtracks*) when it finds that the candidate cannot possibly be completed to a valid solution. It is suitable for those problems which admit the concept of a *partial candidate solution* and quick validity test exists for the partial solution. It is actually an optimization of the brute force technique. In the brute force technique, we exhaustively examine all the combinations in the decision space and evaluate them in the end to find which combination is correct. In backtracking, we build a decision tree and make the decisions incrementally, that is, one variable at a time. It is possible to judge whether this choice will ultimately lead us to a solution or not. If it cannot, then we will not further investigate that branch. We will go back to the previous point in decision making and make another choice. This considerably reduces the number of trials.

### Structure

We will study a set of problems that are suitable for backtracking. We will cover the topics in the following order:

- Backtracking principle
- Question 53: How will you search a word in a grid?
- Question 54: How will you traverse a maze?
- Question 55: How will you find all the combinations of  $n$  numbers taken  $k$  at a time?
- Question 56: How will you partition a string into palindrome segments?

- Question 57: What is the sum of elements of the BST between a range?
- Question 58: How will you partition a set into  $k$  subsets having equal sum?

## Objectives

After studying this unit, you will be familiar with the backtracking principle. You will be able to recognize problems which are suitable for backtracking modeling. You will be confident about solving problems on backtracking.

### 7.1 Backtracking principle

A problem is suitable for backtracking approach if:

- The solution can be built step-by-step by taking decisions at every step.
- There are constraints imposed on decision variables which enable us to identify promising and non-promising decisions.
- We are allowed to do experimentation with a temporarily chosen decision and it is possible to revert back to the original position if the decision was wrong.

The choices at every step constitute a decision tree, called the **state-space tree**. The root of the tree depicts the choices available in the initial state. The nodes at each level represent the choices made for the corresponding component of a solution. This tree is constructed in a DFS manner so that it is easy to look into.

It is different from the exhaustive or brute force approach. In the exhaustive approach, the entire state-space tree is traversed and the validity of solution is checked in the end. In the backtracking algorithm, we build the solution part-by-part. We are able to deduce whether the decision is likely to lead us to the solution. If not, then we will abandon that branch of the state-space tree. Thus, it is an efficient way of implementing the brute force method.

The objectives for backtracking problem fall into three categories.

- **Decision:** Finding any feasible solution subject to the constraints. Here we stop after getting the first complete solution.
- **Enumeration:** Finding all the feasible solutions subject to the constraints. The backtracking algorithm can continue even after reaching a complete solution to find all the other solutions.
- **Optimization:** Here some objective function is specified which is a function of the decision variables. The goal is to optimize (maximize or minimize) the objective function among the feasible solutions. When the goal is finding an optimum solution rather enumerating all solutions, this algorithm is called the *branch and bound* technique. The traveling salesman problem is a typical application of the branch and bound. This problem finds a minimum cost path for the salesman to visit a set of cities and return back to the headquarters.

### Some famous backtracking problems

These problems have been widely documented in various text books, so we will not go into their details. We will just present the problems.

### Place N-Queens

In this problem, we are given an  $N \times N$  chessboard – not necessarily the standard 8 x 8. Our task is to place the N-queens, such that no two queens attack each other, that is, they should not be in the same row or in the same column or on the same diagonal. Consider the brute force method applied to an 8 X 8 board. The number of ways in which the 8 queens can be placed on 64 squares is  ${}^{64}C_8 = 4.4$  billion. We can solve the problem by adding 1 queen at a time and backtracking if we cannot complete the problem of placing 8 queens. There are 92 distinct solutions to the problem. But if you count the rotations and reflections to be the same solution, then there are 12 distinct solutions.

### Find a Hamiltonian circuit in the graph

A Hamiltonian circuit in a graph is the path that starts and ends at the same vertex, and passes through all the other vertices exactly once. We will start from the root vertex and go on visiting the neighbors. If we encounter a

previously visited vertex, we will abandon that path and go back to the previous vertex.

## **7.2 Question 53 – How will you search a word in a grid?**

This is a classic application of the backtracking algorithm. Here we want to search a string on a board containing alphabets. You can traverse the board in the left, right, up, or down directions. At each stage, you have four decisions to make. If you follow the brute force approach, the decision tree will have four branches at the root node. Each of these branches will have 4 more branches and so on till we hit the board boundaries. The brute force approach will traverse the entire decision tree and evaluate the outcome at the end. But in the backtracking method, we try out various choices and abandon the unsuccessful ones.

### **Problem statement**

You are given a board, that is, a square matrix consisting of some letters and a target word. Your task is to find if the word exists in the matrix by traversing the board along the adjacent cells. Here, the *adjacent* cells are those that are horizontally or vertically neighboring. The one-letter cell may be used only once. Let us illustrate this with the help of an example.

#### **Example:**

```
Input: board = [
    ['A', 'B', 'C', 'X'],
    ['S', 'Z', 'C', 'S'],
    ['P', 'D', 'E', 'E']
]
Word = "ABCCED"
```

#### **Output:** True

**Explanation:** We can traverse the matrix starting from cell (0, 0) like this.

A → B → C ↓ C ↓ E ← D

Since the word can be found this way, the output is True.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def exist(self, board: List[List[str]], word: str) ->  
    bool:
```

## Strategy

We can solve the problem by taking one step at a time, and we can retrace if the step proves to be wrong. So this problem is suitable for backtracking. But first we have to find a starting point, that is, the root of the state-space tree. This is nothing but locating the first letter of the target word on the board. We will run a nested loop to scan the rows and columns of the board.

When the first letter is found, we will explore the path by a depth-first search. We will write a recursive function `dfs` for this purpose. If the `dfs` function returns `True`, we will return true from `exist` function. Otherwise, we will continue scanning and returning false if we don't get the starting letter anywhere in the board.

Now let us discuss the `dfs` function. The variables `board` and the `words` are passed as arguments because they are local variables of the `exist` function. In addition, it receives the search coordinates (`x`, `y`) and the currently found word `cur`. The variable `cur` is an empty string in the first call to `dfs` from the `exist` function. If `x`, `y` are going out of bounds or if the character at the `board[x][y]` is blank, then we will return `False`. If everything is OK, we will append the `board[x][y]` to `cur` and see whether it matches the desired word. If it does, then we have found the answer. We will return `True`.

Otherwise, we will prepare for backtracking. We will save `board[x][y]` in a temporary variable and convert `board[x][y]` to a blank, to mark it as visited. Then we will call the `dfs` four times for searching bottom, top, right and left neighbors. If any one of them succeeds in getting the next characters of the word, then the function returns `True`. If all the four attempts fail, then we will do backtracking, that is, to restore the value of at `board[x][y]` from `temp` and return `False`, so that the calling function abandons this attempt and tries another element.

## Python code

The following code implements the strategy described above:

```
1. from typing import List
2. class Solution:
3.     def solution(self, board, word, x, y, cur):
4.         if(x < 0 or x >= len(board)
5.          or y < 0
6.          or y >= len(board[x]))
7.          or board[x][y] == ' '):
8.             return False
9.         cur += board[x][y]
10.
11.        if(len(cur) > len(word)):
12.            return False
13.        if(board[x][y] != word[len(cur)-1]):
14.            return False
15.        if(cur == word): return True
16.        temp = board[x][y]
17.        board[x][y] = ' '
18.
19.        if(self.solution(board, word, x, y+1, cur)): return
True
20.        if(self.solution(board, word, x, y-1, cur)): return
True
21.        if(self.solution(board, word, x+1, y, cur)): return
True
22.        if(self.solution(board, word, x-1, y, cur)): return
True
23.
24.        board[x][y] = temp
25.        return False
26.
```

```

27.     def exist(self, board: List[List[str]], word: str) ->
bool:
28.         if(len(word) == 0):
29.             return True
30.         n = len(board)
31.         for i in range(n):
32.             m = len(board[i])
33.             for j in range(m):
34.                 if(word[0] == board[i][j]
35.                     and self.solution(board, word, i, j,
"")):
36.                     return True
37.         return False
38. temp =\
39. [
40.     ['A','B','C','X'],
41.     ['S','Z','C','S'],
42.     ['P','D','E','E']
43. ]
44. sol=Solution()
45. board=temp[:];print(sol.exist(board,"ABCED"))

```

**Output:** True

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

If the board dimensions are  $m \times n$  and the length of the search word is  $k$ , then the outer loop runs  $mn$  times, and the dfs runs  $k$  times. Hence, the time complexity is  $O(mnk)$ .

### **Space complexity:**

Due to the stack memory requirement, the space complexity is  $O(k)$ .

## 7.3 Question 54 – How will you traverse a maze?

This question is similar to the previous one. Here, instead of an alphabet board, we are required to traverse a maze. In this simple version, you can move around the maze in only two directions. There are other versions where you can move in more directions, but this version is enough to give you the basic idea.

### Problem statement

You are given an  $n \times n$  matrix consisting of 0s and 1s that represent a maze. 0 means that cell is blocked, and 1 means you can step into it. You are initially standing on the top left corner, that is, cell  $(0, 0)$ . Your target is to reach the bottom right corner, that is, cell  $(n-1, n-1)$ . You can move only in two directions, forward and down. Return a matrix showing the path taken with 1s.

Let us illustrate this with the help of an example.

#### **Example:**

#### **Input:**

```
maze = [[1, 1, 1, 1],  
        [1, 1, 0, 1],  
        [0, 1, 1, 0],  
        [1, 1, 1, 1]]
```

#### **Output:**

```
Path = [1, 0, 0, 0]  
       [1, 1, 0, 0]  
       [0, 1, 0, 0]  
       [0, 1, 1, 1]
```

**Explanation:** The output matrix shows the path taken by marking the cells as 1s, that is,  $(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3)$ .

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def solveMaze(self, maze:List[List[int]])->List[List[int]]:
```

## Strategy

We will first create a nested helper function `isSafe(maze, x, y)`. It is a convenient way of checking whether the cells at the coordinates  $(x, y)$  are within the maze limits or not, and that `board(x, y)` is 1. Then we will create an  $n \times n$  matrix named `path` to represent the answer. It will be initialized with 0s. Then we will create the nested recursive function `solveMazeUtil(maze, x, y, path)`. The starting point is given as  $(0,0)$ , so we don't have to search it like the last question. The first call is `solveMazeUtil(maze, 0, 0, path)`. In this function, we will first check the terminating condition, that is,  $(x, y) = ((n-1), (n-1))$ . If it is not met, then we will try setting the `path[x][y]` to 1 and recursively call `solveMazeUtil` with  $x$ ,  $y$  coordinates of the right and down neighbors. If both these calls return False, then we will backtrack, that is, restore the `path[x][y]` to 0 and return False. If any one of the calls succeeds, then the function proceeds to the next level of recursion. If all calls return False, then we finally return False.

## Python code

The following code implements the strategy described above:

```
1. from typing import List  
2. class Solution:  
3.     def solveMaze(self, maze:List[List[int]])->List[int]:  
#Main function  
4. #####Define other nested functions  
5.         # A utility function to check if x, y is valid  
6.         def isSafe(maze, x, y):  
7.             if x >= 0 and x < N and y >= 0 and y < N and  
maze[x][y] == 1:  
8.                 return True
```

```

9.             return False
10.            ##### A recursive utility function to solve Maze
problem
11.            def solveMazeUtil(maze, x, y, path):
12.                # if (x, y) is goal, return True
13.                if x == N - 1 and y == N - 1 and maze[x][y]==
1:
14.                    path[x][y] = 1 #Mark x,y as part of path
15.                    return True
16.                # Check if maze[x][y] is valid
17.                if isSafe(maze, x, y):
18.                    path[x][y] = 1 #Temporarily add x,y to path
19.                    # Try right and down neighbours
20.                    if solveMazeUtil(maze, x + 1, y,
path):return True
21.                    if solveMazeUtil(maze, x, y + 1,
path):return True
22.                    # If none of the above movements work then
23.                    # BACKTRACK: unmark x, y as part of
solution path
24.                    path[x][y] = 0
25.                    return False
26. ######Restart main
27.            N = len(maze)
28.            # Creating a N * N matrix for solution path
29.            path = [[0 for j in range(N)] for i in range(N)]
30.            if solveMazeUtil(maze, 0, 0, path) :return path
31.            return None
32. # Driver code
33. maze = [[1, 1, 1, 1],
34.          [1, 1, 0, 1],
35.          [0, 1, 1, 0],
36.          [1, 1, 1, 1]]
37. sol=solution()

```

```
38. Path = sol.solveMaze(maze)
39. for row in Path:print(row)
```

### **Output:**

```
Path = [1, 0, 0, 0]
[1, 1, 0, 0]
[0, 1, 0, 0]
[0, 1, 1, 1]
```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

There are  $n^2$  cells on the board and at each cell, we spawn two of the recursive calls. So, the worst-case time complexity is  $O(2n^2)$ .

### **Space complexity:**

Due to  $n \times n$  output matrix, the space complexity is  $O(n^2)$ .

## **7.4 Question 55 – How will you find all the combinations of $n$ numbers taken $k$ at a time?**

This is an application of backtracking for enumeration. If we choose  $k$  items out of  $n$  available items, there can be  ${}^nC_k$  combinations. We want to enumerate them all.

### **Problem statement**

You are given two integers  $n$  and  $k$ . Return all the possible combinations of length  $k$  made from the numbers 1.. $n$ . The numbers should not be repeated. Let us illustrate with the help of an example.

### **Example:**

**Input:**  $n = 4$ ,  $k = 3$

**Output:**  $[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]$

**Explanation:** We have to list all the possible combinations of 3 numbers where the numbers should be taken from the list 1, 2, 3, 4.

## Solution format

Your solution should be in the following format:

```
class Solution:  
    def combine(self, n: int, k: int) -> List[List[int]]:
```

## Strategy

The problem is to enumerate all the possible lists of numbers subject to conditions that the length should be  $k$  and the list members are in the range  $1..n$ . To solve this, we will create an empty list named `partial` and add one number at a time to this list.

For this purpose, we will create a nested helper function `dfs(partial, index)`. The first argument is the partial combination list and the second is the index in the range  $1..n$ . The index represents the first number to be appended to the partial. In each step, we append numbers  $\text{index}..n$ . The function first checks the exit condition, that is, whether the length of `partial` has become  $k$ . If yes, then it appends `partial` to the class variable `res(meaning result)` and returns to the previous level of recursion. In effect, it discards the last addition to the `partial` and backtracks one level. If the `partial`'s length is less than  $k$  then we will run a loop where  $i$  goes from `index` to  $n$ . For each value of  $i$ , we will recursively call `dfs(partial + [i], i+1)`. Thus, `partial` list gets appended by  $I$ , and the index for the next iteration is incremented. This continues till  $i$  reaches the max limit of  $n$ . At this point, the program returns to the previous level of recursion. This return statement is not explicitly seen in the code. The return happens at the end of the `for` loop.

Let us walk through the example code. In the first call to the DFS, `partial = []` and `index = 1`. We start a loop where  $i$  goes from the `index` to  $n$ , that is, from 1 to 4 in this case. Before the loop can go any further, the DFS is recursively called with the parameters `partial = [1]` and, `index = 2`, then with `partial = [1, 2]` and `index = 3` and again with `partial = [1, 2,`

`3]` and `index = 4`. Since the partial with a length 3 is built, we will append it to the result `res` and return to the previous level of recursion. This is nothing but backtracking, because `partial` is now `[1, 2]` and `i` is 4. We will call the DFS with parameters `partial = [1, 2, 4]` and `index = 5`. Since the length 3 is achieved, we will append `[1, 2, 4]` to `res` and return to the previous level of recursion. Now, the partial is back to `[1, 2]`, and the index is back to 3. As `i` has reached 4, the function returns to the previous level of recursion. Partial is now back to `[1]` and `i` is 3. It calls the DFS with parameters `partial = [1, 3]` and `index = 4`. Since the length of 3 is yet to be achieved, the DFS gets called once more and the partial becomes `[1, 3, 4]`. It is appended to `res`. In this manner, all the combinations are generated and appended to `res`.

## Python code

The following code implements the strategy described above:

```
1. from typing import List
2. class Solution:
3.     def combine(self, n: int, k: int) -> List[List[int]]:
4.         res = []
5.         #Nested helper function
6.         def dfs(partial, index):
7.             print("On entry
partial=",partial,"index=",index)
8.             if len(partial) == k:
9.                 print("Appended ",partial,index)
10.                res.append(partial)
11.                return
12.            #Call dfs for all values of i
13.            #Backtrace to same after return from dfs
14.            for i in range(index, n+1):
15.                print("Before
",partial,"i=",i,"Ind=",index)
16.                dfs(partial + [i], i+1)
```

```

17.             print("After ",partial,"i=",i,"Ind=",index)
18.             print("Returning")
19.         #Resume main function
20.         dfs([], 1)
21.
22.     return res
23. #Driver code
24. sol=Solution()
25. print(sol.combine(4,3))

```

### **Output:**

`[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]`

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

There are  $n!$  combinations of  $n$  numbers. So, the `dfs` will get called  $n!$  times. So, the time complexity is  $O(n!)$ .

### **Space complexity:**

Due to the stack requirement of  $n!$  recursive calls, the space complexity is  $O(n!)$ .

## **7.5 Question 56 – How will you partition a string into palindrome segments?**

This question demonstrates another type of enumeration using backtracking.

### **Problem statement**

Write a function that partitions a given string  $s$  such that every partition is a palindrome. Return all the possible palindrome partitioning of  $s$ . Remember that a single letter is a palindrome and two identical letters like “aa” are also a palindrome. Let us illustrate with the help of two examples.

### **Example 1:**

**Input:** "abc"

**Output:**

```
[[ "a", "b", "c"]]
```

**Explanation:** Since a single letter is a palindrome, one trivial output is to split all the letters into single letter partitions. In this case, all the letters are distinct, so this is the only output.

### **Example 2:**

**Input:** "aab"

**Output:**

```
[[ "aa", "b"]
 [ "a", "a", "b"]]
```

**Explanation:** In addition to the trivial output, we also have another output where a bigger partition "aa" exists.

### **Example 3:**

**Input:** "ababa"

**Output:**

```
[['a', 'b', 'a', 'b', 'a'], ['a', 'b', 'aba'], ['a', 'bab',
 'a'], ['aba', 'b', 'a'], ['ababa']]
```

**Explanation:** All the possible palindromic partitions are listed.

## **Solution format**

Your solution should be in the following format:

```
1. class Solution:
2.     def partition(self, s: str) -> List[List[str]]:
```

## **Strategy**

We will make use of the DFS, recursion, and the backtracking methodologies. Let us start with a helper function `isPalin` which returns

True if the string passed to it is a palindrome. Then we will write a recursive function `dfs` which is the main workhorse of the algorithm. It will accept a string `s` as a parameter. It will break the string into 1-letter segments, then 2-letter segments, and so on. For this purpose, we will run a loop with `i` going from 1 to `n` (the length of the string). We will pick up a segment consisting of the first `i` characters of `s`, that is, `s[0:i]`, and test if it is a palindrome. In the first iteration, `i=1`. So, we will pick up only 1 letter at a time. If the segment is a palindrome we will append it to a temporary variable. Then we will remove the segment from the string and call `dfs` with the shortened string. When the string reduces to a null string, we will append all the palindromes from the temporary variable to the class variable `res`, and return from the call. After returning, we will clean up the temporary variable to start gathering palindromes for the next run.

Let us walk through example 2 where `s="aab"`. In the first iteration, that is, when `i=1`, `temp` gets `["a", "a", "b"]`. The string `s` starts with `"aab"`, and gets shortened as `"ab"`, `"b"`, and finally `" "`. At this time `["a", "a", "b"]` is appended to the result `res` and the function returns. After returning, there is a backtracking step where the list appended to `temp` is popped back. In our example, the `dfs` was recursively called three times. After `dfs` returns three times, `temp` is again empty and `s` is again `"aab"`. Now `i` becomes 2 and we copy 2 character slices of `s` into `seg`. Now, `seg` becomes `"aa"`. This is a palindrome. So, it is appended to `temp`, and the `dfs` is called again with the first two letters of `s` removed, that is, `s = "b"`. This is again a palindrome, so it is appended to `temp`, and the `dfs` is called again with first two letters of `s` removed. Actually, it has only one letter, so `s = ""`. The strings `["aa", "b"]` are appended to `res` and the function returns from both the recursive calls. Now the string `s` is back to `"aab"` and `temp` is back to empty. Now `i` becomes 3. So, it picks up a 3 letter segment, that is, the full string `"aab"`. Since this is not a palindrome, it is ignored. The variable `i` reaches the limit 3 and the function returns the result `res`, which is `[['a', 'a', 'b'], ['aa', 'b']]`.

## Python code

The following code implements the strategy described above:

```
1. from typing import List
2. class Solution:
3.     def partition(self, s: str) -> List[List[str]]:
4.         self.res = []
5.         self.temp = []
6.         self.dfs(s)
7.         return self.res
8.     def isPalin(self, seg):
9.         i = 0
10.        j = len(seg)-1
11.        while(i < j):
12.            if(seg[i] != seg[j]):
13.                return False
14.            i += 1
15.            j -= 1
16.        return True
17.
18.    def dfs(self, s: str):
19.        res=self.res    #For debugging
20.        temp=self.temp  #For debugging
21.        if(len(s) == 0 and len(self.temp) > 0):
22.            self.res.append(self.temp[:])
23.        return
24.        n = len(s)+1
25.        for i in range(1, n):
26.            seg = s[0:i]
27.            temp=self.temp  #For debugging
28.            if(self.isPalin(seg)):
29.                self.temp.append(seg)
30.                self.dfs(s[i:])
31.                self.temp.pop()
32.            pass
33. #Driver code
34. sol = Solution()
```

```
35. print(sol.partition("aab"))
```

### Output:

```
[['a', 'a', 'b'], ['aa', 'b']]
```

## Complexity Analysis

The complexity analysis is as follows:

### Time complexity:

Let the length of the string  $s$  be  $n$ . Partitioning means putting a separator “|” at  $n-1$  possible places. So, we have a choice of putting or not putting the separator at  $n-1$  places. That amounts to  $2^{n-1}$  choices. For each partition (worst case length  $n$ ), we have to test it for being a palindrome. So, the time complexity is  $O(n2^{n-1})$ , simplified as  $O(n2^n)$ .

### Space complexity:

Since the temp array could grow to  $n$ , and the res array could grow to  $n^2$ . Hence, the space complexity is  $O(n^2)$ .

## 7.6 Question 57 – What is the sum of the elements of the BST within a range?

This is a variation of Question 40, “What is the  $K^{\text{th}}$  smallest element in a binary search tree?” Recall that the **BST** (**binary search tree**) has a property that all the nodes in the left subtree of any given node are less than the given node and all the nodes in the right subtree of that given node are greater than the given node. When a binary search tree is traversed by the in-order method, the nodes are visited in a sorted ascending order.

### Problem statement

You are given the root node of a binary search tree, and two limits, L and R. Return the sum of the values of all the nodes whose values lie between L and R (inclusive). You may assume that the binary search tree has unique values. Let us illustrate with the help of an example.

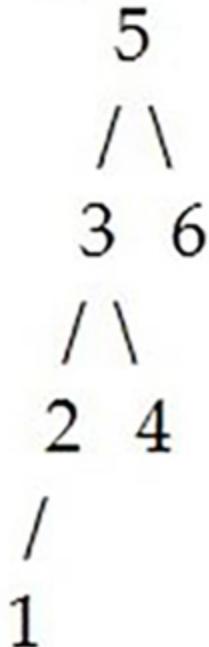
### Example:

```
Root = [5, 3, 6, 2, 4, None, 7], L= 2, R= 6
```

### Output:

```
5
3 6
2 4 None 7
Range sum = 20
```

**Explanation:** The input tree is shown in [Figure 7.1](#):



*Figure 7.1: Example tree*

If we do in-order traversal, the values are 1, 2, 3, 4, 5, 6. If we sum up the values between 2 and 6, the answer is  $2+3+4+5+6 = 20$ .

### Solution format

Your solution should be in the following format:

```
1. class Solution:
2.     def rangeSumBST(self, root: TreeNode, L: int, R: int) -> int:
```

## Strategy

We will first initialize the sum to 0. Then we will traverse the tree in an in-order fashion. We will add the elements traversed to the sum when they are in the range L to R.

The main job is done by the nested function `printInorder`, which traverses the tree in LNR (the left child, the node, the right child) sequence. It recursively calls itself to get the left child, then adds the current node value to the sum if it is within the range L and R and then again calls itself to get the right child. Recursion ends when it tries to go beyond a leaf node and the root is found to be `None`.

## Python code

The following code implements the strategy described above. The driver code from question 40 has been reused:

```
1. from typing import List
2. class TreeNode:
3.     def __init__(self, x):
4.         self.val = x
5.         self.left = None
6.         self.right = None
7.
8. class BinTree:
9.     def printTree(self, root:TreeNode)->None:
10.         LevelList = [root]
11.         self.printLevel(LevelList)
12.     def printLevel(self, LevelList:List[TreeNode])->
List[TreeNode]:
13.         LevelStr = ""
14.         outList = []
15.         ListEmpty = True
16.         for node in LevelList:
17.             if node is None:
18.                 LevelStr += "None "
```

```
19.         outList.append(None)
20.         outList.append(None)
21.     else:
22.         LevelStr += (str(node.val) + " ")
23.         outList.append(node.left)
24.         outList.append(node.right)
25.         ListEmpty = False
26.     if not ListEmpty:
27.         print(LevelStr)
28.         self.printLevel(outList)
29.
30. class Solution:
31.     def rangeSumBST(self, root: TreeNode, L: int, R: int) -> int:
32.         self.sum=0
33.         def printInorder(root):
34.             if root:
35.                 printInorder(root.left) #Recursively call left child
36.                 if root.val:
37.                     if L <= root.val <= R:self.sum +=
38.                         root.val
39.                     printInorder(root.right) #Recursively call right child
40.             printInorder(root)
41.             return self.sum
42.
43. root = TreeNode(5)
44. root.left = TreeNode(3)
45. root.right = TreeNode(6)
46. root.left.left = TreeNode(2)
47. root.left.right = TreeNode(4)
48. root.right.left = TreeNode(None)
```

```
49. root.right.right = TreeNode(7)
50. bst = BinTree()
51. bst.printTree(root)
52. sol = Solution()
53. print("Range sum =", sol.rangeSumBST(root, 2, 6))
```

### **Output:**

```
5
3 6
2 4 None 7
Range sum = 20
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

Since we are traversing the tree once, the time complexity is  $O(n)$ .

### **Space complexity:**

Due to the recursion, we are using a stack of  $n$  elements, and the sorted array of  $n$  elements. So, the space complexity is  $O(n)$ .

## 7.7 Question 58 – How will you partition a set into $k$ subsets having equal sum?

This is a classic example where the backtracking algorithm can be successfully used.

### Problem statement

You are given an array of integers named `nums` and a positive integer  $k$ . Find if we can divide this array into  $k$  non-empty subsets whose sums are all equal. Return `True` if yes. Also, print the resulting subsets. Let us illustrate with the help of an example.

### **Example:**

**Input:** `nums = [1,9,1,1,8,2,5,3,1,3,6,3,3,4]`,  $k=5$

## **Output:**

```
[[9, 1], [8, 2], [6, 4], [5, 3, 1, 1], [3, 3, 3, 1]]  
True
```

**Explanation:** There are five subsets. Sum of elements in each subset is 10.

## **Solution format**

Your solution should be in the following format:

```
1. class Solution(object):  
2.     def canPartitionKSubsets(self, nums, k):
```

## **Strategy**

We will first do some preliminary checks on the given array. We will find the sum of all the numbers in `nums`. If we are to succeed in  $k$  way equal sum partitioning, then each partition sum should be  $\text{sum}/k$ . Thus,  $\text{sum}/k$  must be an integer, and it will be the target for the partition sum. If the division leaves a remainder, then we will immediately return `False`. Then we will sort the array and see if the highest number is greater than the target. If so, then it is impossible to form a group with sum equal to target. Here also, we will return `false`.

Having taken care of the base cases, let us turn to the main algorithm. As we need to create  $k$  groups, we will keep track of the sum of the numbers in each group in an array named `groups`. Initially, all the elements will all be 0. Also, we will create a list of  $k$  empty lists named `sets` to store the members of each group.

We will write a nested recursive function `search(groups)` for the task. The main idea is that we will pop a number from the `nums` one-by-one and insert it in one of the groups. We can't objectively decide which group, so we will just take a "try and backtrack" approach. We will serially scan the `groups` array and insert the popped number in the first group that can accommodate it without its sum exceeding the target. Due to popping, the `nums` array becomes shorter by 1. Now, recursively call `search`. If the choice of the group was correct, recursion will progress till `nums` array eventually

becomes null. In that case, we have found the answer. Otherwise, do backtracking, that is, remove the popped number from the trial group and insert it in the next available group. If all groups are scanned and still no success, then we will return False.

Now let us walk through the code of the example. The array [1, 9, 1, 1, 8, 2, 5, 3, 1, 3, 6, 3, 3, 4] has to be divided into five groups. The sum of the elements is 50. It is divisible by 5. So, the target sum of each group is  $50/5 = 10$ . The nums array gets sorted as [1, 1, 1, 1, 2, 3, 3, 3, 4, 5, 6, 8, 9]. The top element 9 is popped into v. As all the groups are empty, it is inserted into the first group and also in the first set. So the array groups become [9, 0, 0, 0, 0], and the sets become [[9], [], [], [], []]. In the next recursive call, the first group cannot accommodate the next popped element 8. So, it is inserted in the next group. At the end of 4 recursive calls, the groups = [9, 8, 6, 5, 0] and the sets = [[9], [8], [6], [5], []]. The next popped element 4 can be accommodated in groups[2]. So, we have:

```
groups = [9, 8, 10, 5, 0], and sets = [[9], [8], [6, 4], [5], []]
```

This way all the groups are formed. The progress of nums and groups is shown below:

Iter	Nums	Groups
0	[1, 1, 1, 1, 2, 3, 3, 3, 3, 4, 5, 6, 8, 9]	[0, 0, 0, 0, 0]
1	[1, 1, 1, 1, 2, 3, 3, 3, 3, 4, 5, 6, 8]	[9, 0, 0, 0, 0]
2	[1, 1, 1, 1, 2, 3, 3, 3, 3, 4, 5, 6]	[9, 8, 0, 0, 0]
3	[1, 1, 1, 1, 2, 3, 3, 3, 3, 4, 5]	[9, 8, 6, 0, 0]
4	[1, 1, 1, 1, 2, 3, 3, 3, 3, 4]	[9, 8, 6, 5, 0]
5	[1, 1, 1, 1, 2, 3, 3, 3, 3]	[9, 8, 10, 5, 0]
6	[1, 1, 1, 1, 2, 3, 3, 3]	[9, 8, 10, 8, 0]
7	[1, 1, 1, 1, 2, 3, 3]	[9, 8, 10, 8, 3]
8	[1, 1, 1, 1, 2, 3]	[9, 8, 10, 8, 6]

9	[1, 1, 1, 1, 2]	[9, 8, 10, 8, 9]
10	[1, 1, 1, 1]	[9, 10, 10, 8, 9]
11	[1, 1, 1]	[10, 10, 10, 8, 9]
12	[1, 1]	[10, 10, 10, 9, 9]
13	[1]	[10, 10, 10, 10, 9]
14	[]	[10, 10, 10, 10, 10]

**Table 7.1:** Progress of variables *nums* and *groups*

## Python code

The following code implements the strategy described:

```

23.         return False
24.     #Resume main
25.     nums.sort()
26.     if nums[-1] > target: return False
27.     if search([0] * k):
28.         print(sets)
29.         return True
30.     else: return False
31. #Driver code
32. sol = Solution()
33. #nums = [1, 1, 3, 2, 2]
34. #print(sol.canPartitionKSubsets(nums, 3))
35. #nums = [4, 3, 2, 3, 5, 2, 1]
36. #print(sol.canPartitionKSubsets(nums, 4))
37. nums = [1, 9, 1, 1, 8, 2, 5, 3, 1, 3, 6, 3, 3, 4]
38. print(sol.canPartitionKSubsets(nums, 5))

```

## **Output:**

```

[[9, 1], [8, 2], [6, 4], [5, 3, 1, 1], [3, 3, 3, 1]]
True

```

## **Complexity Analysis**

The complexity analysis is as follows.

### **Time complexity:**

The key to find the time complexity is to find how many times the search function is called. The exact analysis is difficult but we can get an estimate of the upper bound. Initially, the groups array contains 0s and the search is called once. As the elements get filled in groups, search is called more times. In worst case, in the first call, search is called once, in second call, it is called twice, and so on. The groups array has  $k$  elements and by the time all of them become non zero, in worst case, search will be called  $1*2*3*...k$  times, that is,  $k!$  times. There are still  $n-k$  items left in the nums array. For them the loop will iterate  $k$  times, trying to put each element from nums in

each group and checking if it works. The time taken will be  $k*k*k\dots(n-k)$  times, that is,  $k^{n-k}$ . The overall time complexity is  $O(k! * k^{n-k})$ .

### Space complexity:

Due to the recursion, we are using a stack. So, the space complexity is  $O(k! * k^{n-k})$ .

## Conclusion

We started with the description of the backtracking algorithm. Then we covered 6 questions which were suitable for using backtracking. Questions 53 and 54 belong to the category of maze solving. You will be able to map any maze problem to these two. Questions 55 and 56 belong to the category of enumeration. In these questions, after finding a solution, we backtrack and search more solutions. In question 57, we have to traverse a tree with the in-order method. This requires backtracking. In question 58, we have to allocate numbers to the groups so that the target group-sum is achieved. We do this by trial and error, which is nothing but backtracking.

In the next chapter we will study the greedy and divide, and conquer algorithms.

## Points to Remember

- Backtracking is an efficient way of implementing the brute force method, where likely unsuccessful paths are detected early and weeded out.
- Backtracking finds a feasible solution, and branch and bound finds an optimal solution.

## MCQs

1. Backtracking algorithm is implemented by constructing a tree of choices called as?
  - A. State-space tree
  - B. State-chart tree

- C. Node tree
  - D. Backtracking tree
2. What is a node called if it has a possibility of reaching a complete solution?
- A. Non-promising
  - B. Promising
  - C. Succeeding
  - D. Preceding
3. Which technique enumerates a list of promising nodes that could be computed to give the possible solutions of a given problem?
- A. Exhaustive search
  - B. Brute force
  - C. Backtracking
  - D. Divide and conquer
4. Which of the following problems cannot be solved by the backtracking method?
- A. n-queen problem
  - B. subset sum problem
  - C. Hamiltonian circuit problem
  - D. Travelling salesman problem
5. What happens when the backtracking algorithm reaches a complete solution?
- A. It backtracks to the root
  - B. It continues searching for other possible solutions
  - C. It traverses from a different route
  - D. Recursively traverses through the same route

## Answers to MCQs

**Q1:** A

**Q2:** B

**Q3:** C

**Q4:** D

**Q5:** B

## **Questions**

1. What is backtracking?
2. How is backtracking different from branch and bound?

## **Key terms**

Backtracking, Branch and bound, enumeration.

## CHAPTER 8

### Greedy and Divide-and-Conquer Algorithms

In this chapter, we will study about two algorithmic techniques, namely greedy, and the divide-and-conquer.

Greedy methodology builds up a solution piece-by-piece, and during each piece, it chooses the next piece such that it offers the most obvious and immediate benefit. It is suitable for problems where choosing the locally optimal solution ultimately leads to the globally optimal solution. Many practical problems fit into this category. However, there are some situations where you need to do a small local sacrifice to achieve a global optimum. In these situations, the greedy algorithms do not work. A classic example is the knapsack problem explained in question 59. Suppose you are traveling in an airline whose baggage allowance is 15 Kg. You need to choose what items to carry such that the value of the items in the knapsack is maximized. You are given a list of  $n$  items and their weights. We will sort the items by their figure merit given by their price/weight ratio and go on filling the knapsack according to the sorted order till the capacity is not exceeded. Now suppose you filled 5 items and reached a weight of 14 Kg. The next item is 2 Kg in weight. What do you do now? If you are allowed to break an item and take its fraction, then we will break the 6<sup>th</sup> item into two and put 1 Kg in the knapsack. This is known as the fractional knapsack problem and it is suitable for the greedy methodology. In another variation, we are not allowed to break the item, we have to take it whole or leave it. This is called 0-1 knapsack problem. The 0-1 knapsack problem is not suitable for greedy methodology. We need dynamic programming to solve it. It is explained in question 64 in the next chapter.

We have already studied questions based on greedy methodology though we didn't specifically call it such. It was question 4 in [Chapter 1](#), "How many boats are required to save people?" In this chapter we will solve some more problems.

## Structure

We will study a set of problems that are suitable for the greedy, and divide-and-conquer techniques. We will cover the topics in following order:

- Some famous problems on greedy methodology
- Some famous problems on divide-and-conquer methodology
- Question 59: How will you maximize the value in a knapsack?
- Question 60: How will you remove the  $k$  digits from a number to get the smallest number?
- Question 61: Can you provide the correct change to lemonade customers?
- Question 62: How will you use the divide-and-conquer strategy for sorting?
- Question 63: How will you find the  $k$  closest points to the origin?

## Objectives

After studying this unit, you will be familiar with the greedy and divide-and-conquer methodologies. You will be able to recognize problems which are suitable for modeling by these techniques and get practice in solving them.

### 8.1 Some famous problems on greedy methodology

Some standard algorithms based on the greedy methodology are listed below. As these are generally well covered in a typical university syllabus, we will not go into their details here. These problems will help you identify the problems suitable for this methodology.

1. **Kruskal's Minimum Spanning Tree (MST):** In this algorithm, we create a minimum spanning tree for a graph by picking the edges one-by-one. The greedy choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.
2. **Prim's Minimum Spanning Tree**  
[\(https://www.geeksforgeeks.org/prims-algorithm-using-](https://www.geeksforgeeks.org/prims-algorithm-using-)

[priority\\_queue-stl/](#)): In this algorithm, too, we create a minimum spanning tree by picking the edges one-by-one. We maintain two sets: a set of vertices already included in the MST, and another set of vertices not yet included. The greedy choice is to pick the smallest weight edge that connects the two sets.

3. **Dijkstra's Shortest Path** (<https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>): This is a very famous algorithm to find the shortest between the two nodes. The shortest path tree is built-up edge-by-edge. We maintain two sets: a set of vertices already included in the tree, and another set of vertices not yet included. Here also, the greedy choice is to pick the edge that connects the two sets and results in the smallest weight path.
4. **Huffman Coding** (<https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/>): This is a loss-less compression technique. It assigns the variable-length bit codes to different characters. The greedy choice is to assign the least bit length code to the most frequent character.

Another important algorithm design philosophy is *divide-and-conquer*. It splits the problem into smaller parts. Often these parts are further split into still smaller parts till the parts consist of single elements. These parts are solved and then combined to form the total solution. This algorithm works when the time complexity is worse than  $O(n)$ , because the overhead of splitting and merging is compensated by the smaller size of the problem.

## **8.2 Some famous applications of divide-and-conquer strategy**

Some standard algorithms based on the divide-and-conquer methodology are listed below. As these too are generally well covered in a typical university syllabus, we will not go into their details here. These problems will help you to identify the problems suitable for this methodology:

- Merge sort
- Quick sort
- Strassen's algorithm for matrix multiplication

- Karatsuba's algorithm for integer multiplication

## **8.3 Question 59 – How will you maximize the value in a knapsack?**

This is a classic case of a greedy algorithm philosophy.

### **Problem statement**

You have a knapsack with a limited carrying capacity. You want to fill it with selected items from a list of items in such a way that the value of the items in the knapsack is maximized. You are given an array named `wt` that contains the weights of the items. Another array named `val` contains their corresponding values. The knapsack capacity is `capacity`. You are allowed to break an item if the remaining capacity cannot accommodate a full item. Write a function that prints the list of selected items and returns the maximum value obtained. Let us illustrate this with the help of an example.

#### **Example:**

##### **Input:**

```
wt = [10, 40, 20, 30]
val = [60, 40, 100, 120]
capacity = 50
```

##### **Output:**

```
Index = 0 value= 60 Full
Index = 2 value= 100 Full
Index = 3 value = 80.0 Fraction 0.6666666666666666
Maximum value in Knapsack = 240.0
```

**Explanation:** The selected item numbers 0 and 2 are full. Now the remaining capacity is  $50 - 10 - 20 = 20$  Kg. It is not enough to accommodate 30 Kg of item 3. So, break item number 3 and take only the two-thirds of it. So, the total value is  $60 + 100 + 30 * .6666 = 240$ .

### **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def getMaxKnapsack(self,wt, val, capacity):
```

## Strategy

Because we are allowed to break an item to fill the remaining capacity, we can use the greedy algorithm. We will rank the items by their figure of merit, that is, the ratio of their value to their weight. Items like gold and jewelry will rank very high in this scale. To facilitate the sorting and to remember the original numbering for printing the selection list, we will define a class `ItemValue` which has data members-weight, value, cost (figure of merit), and original index. Now the question is, how will you sort these objects of the user defined class `ItemValue`? There are two methods. The first method is to define a lambda function which calculates a key by doing some operations on its data fields. Another method is to add a method named `lt` (meaning less than) to the class. It accepts an object of the same class as an argument and returns True if the self is less than the other object. Here we will adopt the later method and define the function `lt` which compares the cost fields of the operands.

The problem of selecting the items will be solved by selecting one item at a time. Our decision will be based on the greedy philosophy, that is, select an item that has the highest value and the lowest weight. That is why we sorted the items on the basis of merit-value. We will go on filling the knapsack, starting with the highest ranked item. The available capacity will be decremented by the weight of the item added. The variable `totalValue` will be incremented with the value of the item. The original index and value are printed. We will continue adding items till the residual capacity becomes 0. But there may be a catch. When it comes to the last item, it is possible that the room left in the knapsack is not enough to accommodate it. If this happens, we will break the item and add only a fraction of it so that it fills the knapsack exactly.

## Python code

The following code implements the strategy described above:

```
1. class ItemValue:  
2.  
3.     #Class to store items data  
4.     def __init__(self, wt, val, index):  
5.         self.wt = wt          #Weight of item  
6.         self.val = val        #Value of item  
7.         self.index = index   #Index of item in list  
8.         self.cost = val / wt #Figure of merit  
9.  
10.    def __lt__(self, other):  
11.        return self.cost < other.cost #Needed for sort  
function  
12.  
13. # Greedy Approach  
14. class Solution:  
15.     def getMaxKnapsack(self, wt, val, capacity):  
16.         #From the given weight and value arrays,  
17.         #prepare an array of ItemValue objects  
18.         iVal = []  
19.         for i in range(len(wt)):  
20.             iVal.append(ItemValue(wt[i], val[i], i))  
21.         # Sort ItemValue array according to figure of  
merit, max first  
22.         iVal.sort(reverse = True)  
23.         totalValue = 0           #Initialize total  
value to 0  
24.         for i in iVal:  
25.             iwt=i.wt  
26.             ival=i.val  
27.             ind=i.index  
28.             if capacity >= iwt:  
29.                 capacity -= iwt  
30.                 totalValue += ival
```

```

31.             print('Index
=', i.index, 'value=', ival, 'Full')
32.         else
33.             fraction = capacity / iwt
34.             totalValue += ival * fraction
35.             print('Index =', i.index, 'value =',
36.                   ival*fraction, 'Fraction', fraction)
37.             break
38.     return totalValue
39.
40. # Driver Code
41. wt = [10, 40, 20, 30]
42. val = [60, 40, 100, 120]
43. capacity = 50
44. sol=Solution()
45. maxValue = sol.getMaxKnapsack(wt, val, capacity)
46. print("Maximum value in Knapsack =", maxValue)

```

## **Output:**

```

Index = 0 value= 60 Full
Index = 2 value= 100 Full
Index = 3 value = 80.0 Fraction 0.6666666666666666
Maximum value in Knapsack = 240.0

```

## **Complexity Analysis**

The complexity analysis is as follows.

**Time complexity:** If the number of items is  $n$ , the time complexity of sorting is  $O(n \log n)$ . The item choosing loop runs  $n$  times. Hence, the time complexity is  $O(n \log n + n) = O(n \log n)$ .

**Space complexity:** Due to the array memory requirement, the space complexity is  $O(n)$ .

## 8.4 Question 60 – How will you remove k digits from a number to get the smallest number?

This question is a good exercise in applying the greedy algorithm technique.

### Problem statement

You are given an n-letter string representing an n-digit number. How can you remove  $k$  letters from the string so that the resulting number is the smallest possible one. But the output number should not have leading zeroes. Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:** num = “1432219”,  $k = 3$

**Output:** “1219”

**Explanation:** As  $k$  is 3, you are allowed to remove 3 digits. If you remove the digits 4, 3, and 2 then the new number is 1219 which happens to be the smallest.

#### **Example 2:**

**Input:** num = “10200”,  $k = 1$

**Output:** “200”

**Explanation:** If you remove the leading 1, the remaining number is 0200. But this is not valid because the output must not contain leading zeroes.

#### **Example 3:**

**Input:** num = “10”,  $k = 2$

**Output:** “0”

**Explanation:** As  $k$  is equal to the length of the input string, you may remove all the letters and leave an empty string representing “0”.

### Solution format

Your solution should be in the following format:

1. class Solution:

```
2.     def getMaxKnapsack(self,wt, val, capacity):
```

## Strategy

The problem of removing  $k$  digits may be broken into  $k$  problems of removing 1 digit at a time. As per the greedy philosophy, we should aim at the maximum reduction. The crux of the problem is to determine the digit whose removal will result in the maximum reduction. Let us illustrate this with an example. Consider the string “524”. If you remove the first digit, then the remaining number is “24”, and if you remove the second digit, then the remaining number is “54”. The lesson that we learn is that when we compare two consecutive digits, if the first digit is greater, then we should remove it. We will create a sliding window of two digits, initially positioned at the beginning of the string. The window is pointed by the indices `idx` and `idx+1`. If the first digit is greater, then we will delete it from `nums`, decrement `k`, and reset the `idx` to 0. If the first digit is equal to or less than the second digit, then we will increment `idx`, that is, shift the comparison window to the next two digits in the hope of finding a falling sequence. We will keep on shifting the window and repeating the process. But we need to check whether the window has reached the right extreme. In that case, we will delete the second digit, decrement `k`, and reset the `idx` to 0.

Now let us see some minor points in the code. First, we will convert the given string into an array of integers. Then we will initialize `idx` to 0 and run a while loop as long as `k` is greater than 0. Firstly, we will check a condition that there is only one element remaining in `nums` and `k` is still not 0. In that case, we will make `nums[0]=0` and break. Then we will apply the logic described above. In the end, the number array has to be converted back to a string. The following statement joins the numbers to a null string to get the string representation:

```
"".join(str(x) for x in num)
```

To take care of the leading zeroes, we will convert it to an integer and again back to a string thus:

```
str(int("".join(str(x) for x in num)))
```

## Python code

The following code implements the strategy described above:

```
1. class Solution:
2.     def removeKdigits(self, num: str, k: int) -> str:
#greedy??"
3.         num = [int(x) for x in list(num)]
4.         idx = 0
5.         #Do till the delete target k is reached
6.         while k > 0:
7.             #If only one number is left, return "0"
8.             if len(num) == 1:
9.                 num[0] = 0
10.                break
11.            #if first number is bigger,
12.            #delete it, decrement k and make idx=0
13.            if num[idx] > num[idx+1]:
14.                del num[idx]
15.                idx = 0
16.                k -= 1
17.            #If the first number is less than or equal to
second
18.            else
19.                if idx == len(num)-2: #Consider corner
20.                    del num[idx+1]
21.                    idx = 0
22.                    k -= 1
23.                else        #Don't delete yet, increment idx
24.                    idx += 1
25.         return str(int("".join(str(x) for x in num)))
26. #Driver code
27. sol=Solution()
28. num = "1432219"; k = 3
29. #num = "10200"; k = 1
```

```
30. #num = "123"; k = 1  
31. print(sol.removeKdigits(num, k))
```

### Output:

1219

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** The sliding window spans the input string of length  $n$  completely. Hence, the time complexity is  $O(n)$ .

**Space complexity:** Due to the numbers array, the space complexity is  $O(n)$ .

## 8.5 Question 61 – Can you provide the correct change to lemonade customers?

This question is a simple demonstration of the greedy algorithm.

### Problem statement

Suppose you are selling “Nimbu Pani”, that is, lemonade at a stall. Each glass costs Rs. 5. Customers have lined up to buy and they will pay cash with the currency notes of Rs. 5, Rs. 10, and Rs. 20. Each time you have to return the right amount of change. You start the business with a tank full of lemonade but no cash at all, so you can return only those notes that are given by earlier customers. The list of notes that are available with the customers in the queue are given in an array of integers named notes. Return True if you can pay back the proper change to every customer. Let us illustrate this with the help of some examples.

#### Example 1:

**Input:** [5,5,5,10,5, 20]

**Output:** true

**Explanation:** From the first 3 customers, we collect three notes of Rs. 5.

From the fourth customer, we collect a Rs. 10 note and return a Rs. 5 note.

From the fifth customer, we collect a Rs. 5 note.

From the sixth customer, we collect a Rs. 20 note. You can either return 3 notes of Rs. 5 or one of 10, and the other of 5. Choose the second option to minimize the risk of change shortage.

Since all the customers got the correct change, the output is True.

### **Example 2:**

**Input:** [10,10]

**Output:** False

**Explanation:** The first customer offers a note of Rs 10 and we don't have change.

### **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def lemonadeChange(self, notes: List[int]) -> bool:
```

### **Strategy**

The problem of returning the change to  $n$  customers can be divided into  $n$  problems of returning the change to individual customers. The decision at each stage depends on the note offered by the customer, and the stock of Rs. 5, and Rs. 10 notes at hand. When you have a choice, your decision should be guided by the principle of greed, that is, conserving the change. We will not bother about monitoring the stock of Rs. 20 notes because we are never going to return them. The decision process can be summarized as:

- If a customer offers a Rs. 5 note, then keep it and increment its stock.
- If a customer offers a Rs. 10 note, then keep it and increment its stock. Return the Rs. 5 note and decrement its stock.
- If a customer offers a Rs. 20 note, then you have a choice of returning 10+5 or 5+5+5. If both Rs. 10 and Rs. 5 notes are available, then you should prefer the 10+5 combination as it will leave more change in your hands.

### **Python code**

The following code implements the strategy described above:

```
1. from typing import List
2. class Solution:
3.     def lemonadeChange(self, notes: List[int]) -> bool:
4.         five = ten = 0          #Initial number of 5 and 10
notes
5.         for note in notes:   #Scan the array
6.             if note == 5:      #If you get Rs 5 note, keep it
7.                 five += 1    #Increment stock of 5's
8.             elif note == 10:#Suppose you get Rs 10 note
9.                 if not five: return False#If you don't have
stock of Rs 5, false
10.                five -= 1   #Return 5 and decrement its
stock
11.                ten += 1    #Increment stock of 10's
12.            else:          #Suppose you get Rs 20 note
13.                if ten and five:#If you have both 10 and 5
in stock
14.                    ten -= 1   #Return 10
15.                    five -= 1   #Return 5
16.                elif five >= 3: #If you have 3 or more 5's
17.                    five -= 3   #Return 3 5's
18.                else
19.                    return False
20.        return True
21. #Driver code
22. sol = Solution()
23. print(sol.lemonadeChange([5,5,5,10,20]))
```

## Output:

True

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** The notes array of length  $n$  is scanned completely. Hence, the time complexity is  $O(n)$ .

**Space complexity:** There is no extra requirement of memory. So, the space complexity is  $O(1)$ .

## 8.6 Question 62 – How will you use the divide-and-conquer strategy for sorting?

This is a very common application of the divide-and-conquer strategy of problem solving, popularly known as quicksort.

### Problem statement

Sort the given array of numbers `nums` and return the result. Let us illustrate this with the help of an example.

**Example:**

**Input:** arr = [10, 7, 8, 9, 1, 5]

**Output:** arr = [1, 5, 7, 8, 9, 10]

**Explanation:** The output is the sorted version of the input.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def quickSort(self, arr:List[int])->List[int]:
```

### Strategy

The basic idea is to select a pivot element from the array and rearrange the array such that the elements smaller than the pivot are placed before the pivot and the elements larger than the pivot are placed after the pivot. This process is called **partitioning**.

We will create the partitioning function `partition(arr, low, high)` that takes in the low and high limits of the array to be partitioned and returns the

partitioning index  $\text{pi}$ . All the elements placed before  $\text{pi}$  are less than the pivot, and all elements after  $\text{pi}$  are greater than the pivot. The partitioning function sets up two pointers  $i$  and  $j$ .  $i$  is initially set to the left of  $\text{low}$ , and  $j$  is swept from  $\text{low}$  to  $\text{high}$ . We will choose the rightmost element as the pivot. There are other ways of selecting the pivot element. It could be the first element, the last element, or any random element.) Our idea is to move elements less than the pivot to the left side. So if we find an element  $\text{arr}[j]$  which is less than the pivot, then we will increment  $i$  and interchange  $\text{arr}[i]$  and  $\text{arr}[j]$ . When the sweep of  $j$  is complete, all the elements up to  $i$  are less than the pivot. Now we will put the pivot in the next place by interchanging  $\text{arr}[\text{high}]$  with  $\text{arr}[i+1]$ . Thus,  $i+1$  becomes the pivot index  $\text{pi}$ .

The recursive helper function `qs_recur(self, arr, low, high)` controls the program flow. We will call it from the main program as, `qs_recur(arr, 0, n-1)`. It will call the partition function, which will break the array into two parts. The first part is ( $\text{low} \dots \text{pi}-1$ ), and the second part is ( $\text{pi}+1 \dots \text{High}$ ). Hopefully these parts are of equal length. Now we will recursively call `qs_recur` on these parts. This is the divide-and-conquer strategy. Recursion will continue as long as the starting index is less than the ending index.

## Python code

The following code implements the strategy described above:

```
1. from typing import List
2. class Solution:
3.     def quickSort(self, arr:List[int])->List[int]:
4.         n = len(arr)
5.         self.qs_recur(arr, 0, n-1)
6.     def qs_recur(self, arr, low, high):
7.         if low < high:
8.             # recursion continues
9.             # as long as low is less than high
10.             #Partition the array and return partition index
11.             pi = self.partition(arr, low, high)
```

```

12.          # Make two recursive calls for elements
13.          # before pi and after pi
14.          self.qs_recur(arr, low, pi-1)
15.          self.qs_recur(arr, pi+1, high)
16.      # The partition function takes last element as pivot,
places
17.      # the pivot element at its correct position in sorted
18.      # array, and places all smaller elements
19.      # to left of pivot and all greater elements to right
20.      # of pivot
21.      def partition(self,arr,low,high):
22.          i = low-1           # index of smaller element
23.          pivot = arr[high]    # pivot
24.          for j in range(low, high):
25.              # If current element is smaller than the pivot
26.              #arrj=arr[j]
27.              if arr[j] < pivot:
28.                  # increment index of smaller element
29.                  i = i+1
30.                  arr[i],arr[j] = arr[j],arr[i]
31.
32.          arr[i+1],arr[high] = arr[high],arr[i+1]
33.          return (i+1)
34. # Driver code to test above
35. arr = [12, 7, 8, 9, 1, 6]
36. n = len(arr)
37. sol=Solution()
38. sol.quickSort(arr)
39. print ("Sorted array is:",arr)

```

## **Output:**

[1, 6, 7, 8, 9, 12]

## **Complexity Analysis**

The complexity analysis is as follows:

**Time complexity:** The running time heavily depends on the selection of the pivot element. If the pivot divides the array into two nearly equal halves, then the time required is least. It has been shown that the average and the best case time complexity is  $O(n \log n)$ . The worst case happens when the pivot divides the array into one part having a single element and the second part having  $n-1$  elements. This requires  $O(n)$  recursive calls, resulting in the worst case time complexity of  $O(n^2)$ . If we select the pivot element randomly, then the probability of worst case happening becomes very small.

**Space complexity:** Due to the stack requirement for recursion, the space complexity is  $O(n)$ .

## 8.7 Question 63 – How will you find the k closest points to the origin?

This is a variation of the last problem which was based on quicksort.

### Problem statement

You are given a list of  $x, y$  coordinates of points in a plane. Find the  $k$  points that are closest to the origin. Closeness is defined by the Euclidean distance. The order of the  $k$  closest points is immaterial.

Let us illustrate this with the help of an example.

#### **Example:**

**Input:** points = [[3, 3], [5, -1], [-2, 4]], K = 2

**Output:** [[3,3], [-2,4]]

**Explanation:** The distances to the origin are  $\sqrt{3^2 + 3^2} = \sqrt{18}$ ,  $\sqrt{5^2 + 1^2} = \sqrt{26}$  and  $\sqrt{2^2 + 4^2} = \sqrt{20}$ . For comparing, you don't actually have to take the square root. The two closest points are (3, 3) and (-2,4). The answer [[-2,4],[3,3]] is also correct because the order is immaterial.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def kClosest(self, points: List[List[int]], K: int) ->  
List[List[int]]:
```

## Strategy

In a roundabout manner, this is a sorting problem. Instead of giving the items to be sorted directly, we are given point coordinates, and we need to find the Euclidean distance of the points from the origin to sort them. We will consider two approaches for this problem.

## Approach 1 – Sorting

The simplest thought that comes to mind is to calculate the array of Euclidean distances, sort it, and pick up the first  $k$  elements. Python provides the readymade sort function to make the job easy. The sort function takes an optional argument which specifies the function to be used to calculate the sorting key.

## Python code

The following code implements the strategy described:

```
1. class Solution(object):  
2.     def kClosest(self, points, K):  
3.         points.sort(key = lambda P: P[0]**2 + P[1]**2)  
4.         return points[:K]  
5. #Driver code  
6. sol=Solution()  
7. points = [[3,3],[5,-1],[-2,4]]; K = 2  
8. print(sol.kClosest(points,K))
```

## **Output:**

```
[[3, 3], [-2, 4]]
```

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** The time complexity of sort is  $O(n \log n)$ . So, the overall time complexity is  $O(n \log n)$ .

**Space complexity:** There is no extra memory requirement. So, the space complexity is  $O(1)$ .

## Approach 2 – Selection sort

There is a crucial point in the problem statement that the order of the  $K$  closest points is immaterial. This means that we don't have to do the full sorting. It is enough to do a partial sorting, that is, divide the array into two buckets, the first one having values less than a certain pivot value, and the second one having values greater than the pivot value. This technique is called the selection sort and it is a variation of the quicksort that we studied in the previous problem. If we are able to find a pivot such that the size of the first bucket is  $K$ , then we have solved the problem.

We will use a recursive helper function `sort(i, j, K)`, where  $i$  and  $j$  are the left and right limits of the portion of the array that we want to rearrange, and  $K$  is the desired size of the first bucket. Initially, we will call it with values:

```
sort(0, len(points) - 1, K)
```

We will randomly choose an index  $k$  (not to be confused with  $K$ ) lying between  $i$  and  $j$  as a pivot. Our partition function uses the leftmost element as the pivot. So, the `points[k]` will be placed at the left end, that is, at index  $i$ . The original contents of index  $i$  will be moved to index  $k$ . Then, we call the function `partition(i, j)` which rearranges the elements between the indices  $i$  and  $j$ , and returns the index corresponding to the pivot, namely `mid`. The first bucket lies from the index  $i$  to `mid`, and its length is `mid-i+1`. If  $K$  is less than this length, then the bucket is too long. The correct pivot lies somewhere between  $i$  and `mid-1`. We need to find it using the same function. The range of search will now be reduced to  $i...mid-1$ . We will recursively call:

```
sort(i, mid - 1, K)
```

If K is greater than the length of the first bucket, then the required pivot lies in the right bucket, that is, from  $\text{mid}+1$  to  $j$ . The size of the bucket will be  $K - (\text{mid} - i + 1)$ . So, we will recursively call:

```
sort(mid + 1, j, K - (mid - i + 1))
```

If  $K$  is neither smaller nor greater than the mid, then we have arrived at the answer.

Now let us take a look at the `partition(i, j)` function. It receives the left and the right indices,  $i$  and  $j$  respectively as parameters. The value of the array at  $i$  will be taken as pivot. The original value of  $i$  will be saved as  $oi$ . Using the indices  $i$  and  $j$ , we will adopt a pinching window approach to find the elements in the wrong bucket and interchange them. The index  $j$  is properly positioned at the right end. We will initialize the left index to  $oi+1$ . We will not do anything to the pivot element at  $oi$ . We will move it to its right place at the end. We will actually never construct an array of Euclidean distances. We will use the array of points itself but use the lambda function `dist` to compare magnitudes.

Now we will start an infinite loop. In another nested loop, we will examine the element `points[i]`, and if it is less than the pivot, keep incrementing  $i$ . We will stop if an element belonging to the right bucket is encountered or  $i$  has equaled  $j$ . Then we will run another while loop. In this loop, we will examine the element `points[j]`, and if it is greater than the pivot, keep decrementing  $j$ . We will stop if an element belonging to the left bucket is encountered or  $j$  has equaled  $i$ . At this time, if the window is fully pinched, that is,  $i \geq j$ , then we will break the infinite loop, otherwise interchange the elements at indices  $i$  and  $j$  and continue.

When we come out of the loop, all the elements to the right of  $j$  are greater than the pivot, and all elements to the left of  $j$  including  $j$  are less than the pivot. Now we will interchange the elements  $oi$  and  $j$ , and return  $j$  as the new pivot index.

## Python code

The following code implements the strategy described above:

```
1. from typing import List
2. import random
3. class Solution:
4.     def kClosest(self, points: List[List[int]], K: int) ->
List[List[int]]:
5.         dist = lambda i: points[i][0]**2 + points[i][1]**2
6.         def sort(i, j, K):
7.             if i >= j: return
8.             #Select k randomly between i and j
9.             # Put random element as A[i] - this is the pivot
10.            k = random.randint(i, j)
11.            points[i], points[k] = points[k], points[i]
12.            mid = partition(i, j)
13.            if K < mid - i + 1:
14.                sort(i, mid - 1, K)
15.            elif K > mid - i + 1:
16.                sort(mid + 1, j, K - (mid - i + 1))
17.        def partition(i, j):
18.            # Partition by pivot A[i], returning an index
19.            # such that A[i] <= A[mid] <= A[j] for i < mid
20.            # < j.
21.            oi = i
22.            pivot = dist(i)
23.
24.            while True:
25.                di=dist(i)
26.                dj=dist(j)
27.                while i < j and dist(i) < pivot:
28.                    i += 1
29.                while i <= j and dist(j) >= pivot:
30.                    j -= 1
31.                if i >= j: break
```

```

32.             points[i], points[j] = points[j], points[i]
33.
34.             points[oi], points[j] = points[j], points[oi]
35.             return j
36.         #Resume main function
37.         sort(0, len(points) - 1, K)
38.         return points[:K]
39.
40. #Driver code
41. sol=Solution()
42. points = [[3,3],[5,-1],[-2,4]]; K = 2
43. print(sol.kClosest(points,K))

```

## **Output:**

`[[3, 3], [-2, 4]]`

## **Complexity Analysis**

The complexity analysis is as follows:

**Time complexity:** It is difficult to accurately estimate the time complexity because it is heavily dependent on the initial order. The average time complexity is  $O(n)$ , which is definitely better than full sort, but in the worst case it could be  $O(n^2)$ .

**Space complexity:** Due to the stack memory requirement, the space complexity is  $O(n)$ .

## **Conclusion**

In this chapter you were introduced to two important algorithmic techniques, namely greedy, and divide-and-conquer. We studied some famous questions suitable for these techniques so that you can recognize the problems of these categories. We studied 5 questions related to these topics. In question 59, we selected the items to be put in a knapsack by sorting them by the price/weight ratio. The greedy algorithm was chosen because the items could be broken into fractions. In question 60, it was crucial to

determine a greedy way of determining which digits to remove in order to reduce the value of a number. Once we figure that out, we simply apply the greedy algorithm. In question 61, again the key point was to determine the greedy strategy, i.e., retain the maximum change in hand. In question 62, we successively applied the divide-and-conquer strategy for sorting an array. Question 63 was a disguised sorting problem. Here, the key trick in achieving a low time complexity was to use the fact that we were interested in partial sorting.

In the next and the last chapter, we will study a very important algorithmic strategy, dynamic programming.

## **Points to Remember**

- In the greedy algorithm, we divide the problem in various decision steps and at each step, make a choice that optimizes the outcome.
- In the divide-and-conquer methodology, we break the problem into smaller sub-problems and solve them.
- The following statement converts a character list num to a string representation and removes the leading zeros:
  - `str(int("".join(str(x) for x in num)))`

## **MCQs**

1. The fractional knapsack problem is based on
  - A. Dynamic programming
  - B. Branch-and-bound
  - C. Greedy algorithm
  - D. Divide-and-conquer
2. Which of the following algorithms do not return an optimum solution?
  - A. Dynamic programming
  - B. Backtracking
  - C. Branch-and-bound

- D. Greedy
3. The running time of quick sort heavily depends on the
- Order of the elements
  - Selection of the pivot element
  - Number of inputs
  - Size of the elements
4. Which of the following algorithms is divide-and-conquer?
- Bubble sort
  - Insertion sort
  - Quick Sort
  - All of the above
5. In what manner is a state-space tree for a backtracking algorithm constructed?
- Depth-first search
  - Breadth-first search
  - Twice around the tree
  - Nearest neighbor first

## Answers to the MCQs

**Q1:** C

**Q2:** B

**Q3:** B

**Q4:** C

**Q5:** A

## Questions

- Quote on example where the greedy algorithm does not work.
- What is the need of choosing the pivot randomly in quick sorting?

## Key terms

Divide-and-conquer, greedy, quicksort, pivot, knapsack.

# CHAPTER 9

## Dynamic Programming

Dynamic programming is an algorithmic concept which breaks down a problem into smaller sub-problems and solves these sub-problems individually. The solutions of the sub problems are stored in an array (or some kind of data structure), so if a sub-problem is repeated, we can retrieve the answer from the stored data base rather than recalculating it. Finally, the sub-problems are combined to get the final answer.

The dynamic programming encompasses several concepts we studied earlier, like the divide-and-conquer, the recursion, the memoisation, the mathematical induction, the depth-first search, and so on. It was invented by Richard Bellman in the 1950s while working with the RAND Corporation. His employer did not encourage pure mathematical research, so he named it “Dynamic Programming” so that it appeared commercial. The name has stuck since then.

The dynamic programming concept is somewhat difficult to grasp. But there is a general pattern of thinking. Once you understand the pattern and study the 12 questions discussed here, you will be able to map any other question to one of these patterns.

### Structure

First, we will introduce the concept of dynamic programming with the help of the Fibonacci series example. Then, we will solve some problems to get mastery over the technique. We will cover the topics in the following order:

- Principles of dynamic programming
- Question 64: How will you maximize the value in a 0/1 knapsack?
- Question 65: How to maximize the sales of a door-to-door salesman?
- Question 66: What is the best time to buy and sell stock?
- Question 67: What is the best time to buy and sell stock (part 2)?

- Question 68: In how many ways can you climb the stairs?
- Question 69: How will you minimize the cost of the climbing stairs?
- Question 70: What is the minimum number of coins to dispense the change?
- Question 71: How many coin patterns are there to dispense the change?
- Question 72: How many unique paths exist in a square grid?
- Question 73: Unique paths in a square grid having obstacles
- Question 74: How many unique paths exist in a square grid having obstacles?
- Question 75: How much rain water can be trapped in the ridges?

## Objectives

After studying this unit, you will be familiar with the technique of splitting the problem into small steps, establishing recurrence relations, and storing intermediate results. You will be confident of solving the problems involving dynamic programming.

### 9.1 Principles of dynamic programming

For the dynamic programming to be applicable, the problem should have two basic properties:

1. **Optimal substructure property:** This property states that the problem can be broken into smaller sub-problems, and that the optimal solution of the given problem can be obtained by combining the optimal solutions of its sub-problems.

For example, let us consider the problem of finding the shortest path from node  $u$  to node  $v$  in a graph. The path passes through various intermediate nodes. Suppose  $x$  is one such intermediate node. If you append the shortest  $x-v$  path to the shortest  $u-x$  path, it results in the shortest  $u-v$  path. The Floyd–Warshall algorithm for finding the shortest path between all the pairs algorithm, and the Bellman–Ford

algorithm for finding the shortest path between a given source and destination are some typical examples of the dynamic programming.

All problems don't have the optimal substructure property. For example, the longest path problem, which is the opposite of the shortest problem doesn't have the optimal substructure property.

2. **Overlapping sub-problems:** Utilizing the above property and the principle of divide-and-conquer, the dynamic programming method splits the problem into sub-problems, and then combines the solutions. The dynamic programming is useful when the solutions of the same sub-problems are needed again and again, that is, there are overlapping sub-problems. To minimize the computation, previous solutions are stored in a table so that these don't have to be computed again.

In the problems of finding the Fibonacci numbers or finding the factorial of a number, we need the solution of the sub-problems again and again.

But consider the problem of finding an element in a sorted array using the binary search. Here as well, we divide the problem into smaller parts but these parts are totally independent. Their results are not required again. So, the dynamic programming is not useful here. The problem of merge sort is another such problem. If the algorithm divides the problem but cannot reuse the results of the sub-problems, it is called divide-and-conquer.

There are four steps in applying the dynamic programming approach to a problem:

1. Break the problem in small steps, and find a recurrence relation between the steps to build a recursive solution.
2. Analyze the solution and look for overlapping sub-problems.
3. Store the intermediate results in some data structure for future use.
4. When you apply memoisation to a recursive solution, you decrease the time complexity, but increase the space complexity. If possible, try to build the bottoms-up solution which can be solved by the iteration, rather than the recursion. It will reduce the space complexity. In some

cases, the bottoms-up approach may naturally come first. In that case, go for it directly.

We will illustrate the procedure with the help of two examples, namely the Fibonacci numbers and the Tribonacci numbers. Let us take the first example.

## Problem statement

Find the  $n^{\text{th}}$  Fibonacci number.

We discussed this problem in *Section 1.10.4* while discussing the measurement of time complexity. Let us apply dynamic programming technique now.

1. The problem of finding  $n^{\text{th}}$  Fibonacci number can be divided into two smaller problems. Find  $(n-1)^{\text{th}}$  Fibonacci number, and find  $(n-2)^{\text{th}}$  Fibonacci number. We can use the recurrence relation as follows:

```
If n<2: F(n) = n      (Base case)  
Else:    F(n) = F(n-1) + F(n-2)
```

This immediately lends itself to a recursive solution.

The Python code in *Section 1.10.4* was as follows:

```
1. def recur_fibo(n):  
2.     if n <= 1:  
3.         return n  
4.     else  
5.         return(recur_fibo(n-1) + recur_fibo(n-2))
```

This code worked but the time complexity was found to be  $O(2^n)$ . We will reduce it using the dynamic programming.

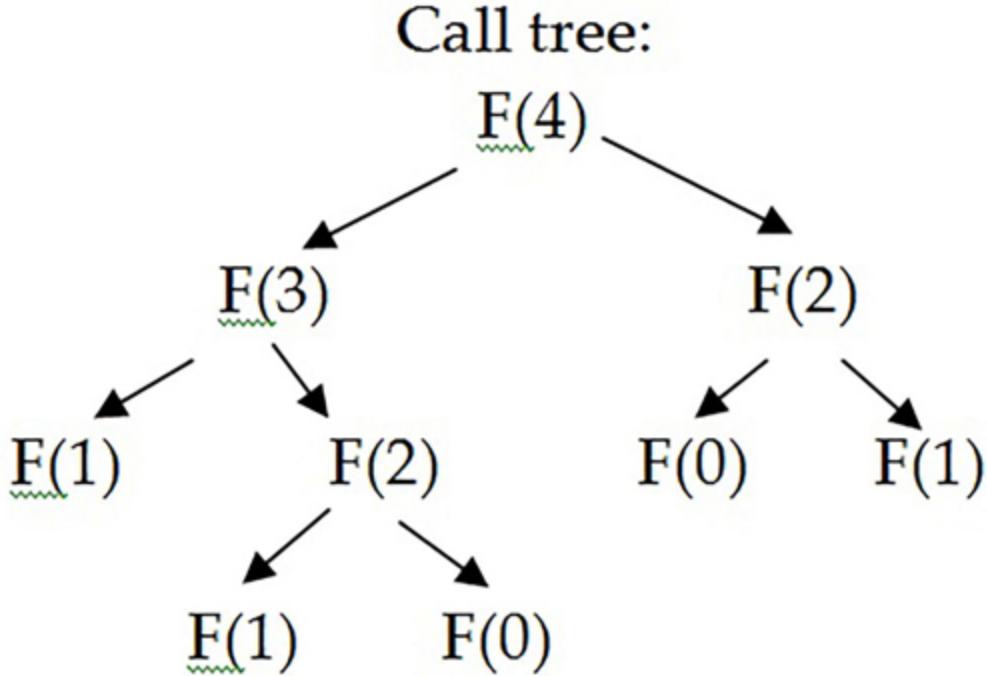
2. We will look for overlapping sub-problems. Notice the calculation sequence for  $n=4$ :

$$\begin{aligned}F(4) &= F(3) + F(2) \\F(3) &= F(2) + F(1) \\F(2) &= F(1) + F(0)\end{aligned}$$

$$F(1) = 1$$

$$F(0) = 0$$

The sequence of the function calling is shown in [Figure 9.1](#):



*Figure 9.1: Call tree of the Fibonacci function*

We observe that the Fibonacci function is called nine times. But also observe that **F(0)** is called two times, **F(1)** is called three times, and **F(2)** is called two times. If we remove these four extra calls, only five calls are left. The repetitions are more glaring for larger values of  $n$ .

3. Let us store the intermediate results in an array, named `FibArray`, so that a repeated call with the same argument can get the results from the array rather than recalculating it. This is called memoisation. Actually, it means memorization of the sub-problem solutions. But it is customary to drop the ‘r’ and call it memoisation.

`FibArray` will initially contain  $n$  zeroes. We will use the memoised function `memo_fibo(n)` to calculate the Fibonacci numbers. The function `memo_fibo(n)` will first check if the `FibArray(n)` is non-zero. If it is non-zero, it means that this has been already calculated. So, it will simply use the value. Otherwise, it will calculate the value

and fill it in the FibArray(n). We know that  $F(0) = 0$  and  $F(1) = 1$ . So, the first two elements of FibArray should be 0 and 1.

## Python code

The following code implements the memorization strategy:

```
1. def memo_fibo(n):
2.
3.     # Taking 1st two fibonacci numbers as 0 and 1
4.     FibArray = [0, 1]
5.
6.     while len(FibArray) <= n :
7.         FibArray.append(0)                      #Append 0 to
initialize it
8.         if n <= 1:                          #Exit condition:
9.             return n                         #We know that F(0) = 0, F(1)
= 1
10.        else
11.            if FibArray[n - 1] == 0:          #F(n-1) done ?
12.                FibArray[n - 1] = memo_fibo(n - 1) #No. Do it
and save result
13.
14.            if FibArray[n - 2] == 0:          #F(n-2)
done ?
15.                FibArray[n - 2] = memo_fibo(n - 2) #No. Do it and
save result
16.
17.        FibArray[n] = FibArray[n - 2] + FibArray[n - 1]
18.        return FibArray[n]
19. #Driver code
20. print(memo_fibo(5))
```

## **Output:**

Since the duplication of work is avoided, the time complexity is reduced to  $O(n)$ . The above solution is an acceptable dynamic programming solution, but the space complexity is still  $O(n)$  due to the use of FibArray and the stack requirement of recursion. We can reduce the space complexity to  $O(1)$  if we build a bottoms-up approach rather than the top-down approach described above, that is, we will calculate the Fibonacci numbers from 0 to  $n$  rather than  $n$  down to 0. Now, we don't have to store  $n$  intermediate results. Observe that the calculation of  $F(n)$  requires only  $F(n-1)$  and  $F(n-2)$ . So, we actually need to store only two values at a time. We will build the solution in an iterative manner rather than a recursive manner. This will avoid the stack requirement and avoid the function calling overhead.

## Python code

The following code implements the iterative approach:

```
1. def memo2_fibo(n):
2.     fn_2 = 0
3.     fn_1 = 1
4.     if n<2:
5.         return n
6.     else:
7.         for i in range(2,n+1):
8.             ans = fn_1 + fn_2
9.             fn_2 = fn_1
10.            fn_1 = ans
11.        return ans
12. #Driver code
13. print(memo2_fibo(9))
```

## **Output:**

34

Let us now turn to the second example.

## Problem statement

Find the  $n^{\text{th}}$  Tribonacci number.

The Tribonacci sequence is similar to the Fibonacci sequence. As the name suggests, it sums the previous three numbers to get the new number. The  $n^{\text{th}}$  number,  $T_n$  is defined as follows:

$$T_n = T_{n-1} + T_{n-2} + T_{n-3}$$

Where  $T_0 = 0$ ,  $T_1 = 1$ ,  $T_2 = 1$ ,  $n$  is assumed to be positive.

## Strategy

The first two steps will be the same as the previous example. But the purpose of this example is to introduce an entirely new way of implementing memoisation. We will use the `@lru_cache` facility from the `functools` module of Python. When you append `@lru_cache` in the beginning of a function, it remembers the parameters and result of the previous calls to the function. Thus, the memoization is automatically achieved. This method is very useful if you want to use the top-down approach.

## Python code

The following code implements this strategy:

```
1. from functools import lru_cache
2. class Solution:
3.     @lru_cache(None) #Store data in cache for next calls
4.     def tribonacci(self, n: int) -> int:
5.         if n==0: return 0
6.         if n<3:  return 1
7.         return self.tribonacci(n-3) + self.tribonacci(n-2) +
self.tribonacci(n-1)
8. #Driver code
9. sol=Solution()
10. print(sol.tribonacci(30))
```

## **Output:**

## 9.2 Question 64 – How will you maximize the value in a 0/1 knapsack?

This is an extension of question 59 of the last chapter. There, we used the greedy algorithm because you were allowed to break an item. Now, we consider a case where you are not allowed to break an item. That is why, it is popularly known as 0/1 knapsack problem. Here, the dynamic programming is required.

### Problem statement

You have a knapsack with a limited carrying capacity. You want to fill it with the selected items from a list of items in such a way that the value of the items in the knapsack is maximized. You are given an array, named `wt` that contains the weights of the items. Another array, named `val` contains their corresponding values. The knapsack capacity is `capacity`. You are not allowed to break an item if the remaining capacity cannot accommodate a full item. Write a function that prints the list of selected items and returns the maximum value obtained. Let us illustrate this with the help of an example.

#### **Example:**

**Input:** `val = [60,50,70,30]`, `wt = [5,3,4,2]`, `W = 5`

**Output:** 80

**Explanation:** If we choose items 1 and 3, the total weight is  $3+2=5$ , and the total value is

$50+30=80$ . This is the best combination.

### Solution format

Your solution should be in the following format:

1. `class Solution:`
2.     `def knapSack(self, w, wt, val):`

## Strategy

The simplest strategy that you can think of is the brute force. Enumerate all the subsets of the given set, and out of those that satisfy the weight constraint, choose the one that gives the maximum value. I think you can work this out on your own. However, this is going to cost you the time complexity of  $O(2^n)$ .

Let us think of a smarter way. We will build a bottoms-up solution because that is easier and natural. Instead of directly going for the problem of filling the 5 Kg knapsack, we will go step-by-step. Starting from 0, at each step, we will increase the capacity by 1 Kg. At each step, we will consider the available item options, that is, only item 0, items 0 and 1, items 0, 1, and 2, and items 0, 1, 2, and 3. At each step, we will keep a track of the best answer. When we reach the capacity of 5 Kg and the choice of four items, we will have our answer. We will create a table as shown in [Table 9.1](#) to store the best value achievable for the future use. We will call this table  $K$  in our program:

Capacity C(n)=w	0 Kg	1 Kg	2 Kg	3 Kg	4 Kg	5 Kg
No item selected						
Item 0: wt=5, val=60						
Item 1: wt=3, val =50						
Item 2: wt=4, val =70						
Item 3: wt=2, val =30						

*Table 9.1: Memoisation table - Initial*

We will fill the table row-wise. Each column represents the available knapsack capacity. Consider the first row ( $i=0$ ) when no item is available for filling. Now, irrespective of the knapsack capacity, the value obtained will be 0.

Now consider the next row, that is,  $i=1$ . The row  $i$  corresponds to item  $i-1$ . Here, only the item 0 is available for filling. The weight of this item is 5 Kg. So we cannot choose this item for capacities less than 5. So, in the

table, the first 5 columns will be 0. and the last column will be 60. To generalize:

$$\begin{aligned} K[1][w] &= 0 && \text{for } w < wt[0] \\ &= val[0] && \text{for } w \geq wt[0] \end{aligned}$$

The table will look as shown in [Table 9.2](#):

Capacity C(n)=w	0 Kg	1 Kg	2 Kg	3 Kg	4 Kg	5 Kg
No item selected	0	0	0	0	0	0
Item 0: wt=5, val=60	0	0	0	0	0	60
Item 1: wt=3, val =50						
Item 2: wt=4, val =70						
Item 3: wt=2, val =30						

*Table 9.2: Memoisation table-Step 1*

Now, let us fill the third row. Here, items 0 and 1 are available for filling. Let us generalize for  $i^{\text{th}}$  row and find a recurrence relation. At every column  $w$ , we have a choice to make.

1. Don't choose the item  $i$ . The value that you can get is the value from the cell above, that is,  $K[i-1][w]$  because that represents the best choice for weight  $w$  when this item was not available.
2. Choose this item if its weight is less than or equal to the knapsack capacity. If you do, then you add  $val[i-1]$  to the answer but at the same time you reduce the capacity available to the other items by  $wt[i-1]$ . The remaining capacity will be  $w-wt[i-1]$ . Don't get confused, row  $i$  represents item  $i-1$ . We have to see what was the best value obtained when the knapsack capacity was  $w-wt[i-1]$  when this item was not available. We will get it at  $K[i-1][w-wt[i-1]]$ . Add this to  $val[i-1]$  to get the total value obtainable from this choice.

Choose the better of the two options. This gives us the recurrence relation to the compute  $K[i][w]$  in terms of earlier computed values.

$$K[i][w] = 0 \quad \text{if } i=0 \text{ or } w=0$$

$$\begin{aligned}
 &= \max(\text{val}[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]) \quad \text{if } wt[i-1] \leq w \\
 &= K[i-1, w] \quad \text{otherwise}
 \end{aligned}$$

By using the above formula, we can fill the table and finally, it looks like what is shown in [Table 9.3](#):

Capacity C(n)=w	0 Kg	1 Kg	2 Kg	3 Kg	4 Kg	5 Kg
No item selected	0	0	0	0	0	0
Item 0: wt=5, val=60	0	0	0	0	0	60
Item 1: wt=3, val =50	0	0	0	50	50	60
Item 2: wt=4, val =70	0	0	0	50	70	70
Item 3: wt=2, val =30	0	0	30	50	70	80

*Table 9.3: Memoisation table - Final*

Our answer is available in  $K[n][w]$ , which is 80.

But we also want to know which items got selected. It is tempting to print the selection when we are finding the max value. But that will not work because the final decision can change due to the future events. So, we have to back trace from the final decision. We will start with  $i = n$ ,  $j = w$ , and  $v = K[n][w]$ .

The rule to recognize a selected item is: for any  $i, j$ , row  $i$  is selected when  $K[i-1][j]$  is less than  $K[i, j]$ . We will keep going up i.e. decrementing  $i$  till we meet the above condition. When met, we will print the item  $i-1$  as the selected item and go back to a smaller problem, where  $i$  is reduced by 1,  $j$  is reduced by  $wt[i-1]$ , and  $v$  is reduced by  $val[i-1]$ . We will continue this till the value becomes 0. In [table 9.3](#),  $K[4,5]$  is 80. It is greater than  $K[3,5]$  which happens to be 50, indicating that the item 4-1, i.e. the item 3 is selected. Now, let us reduce  $V$  by  $val[3]$ , i.e. 30, reduce  $j$  by  $wt[4-1]$  i.e., 2, and reduce  $i$  by 1. We arrive at [3,3], find that  $K[3,3]$  is 50. But  $K[2,3]$  is also 50, indicating that the item 3-1, i.e., the item 2 is not selected. We will go up one cell and find that  $K[1,3]$  is 0. Since this is smaller than  $K[2,3]$ , it indicates that the item 2-1, i.e., the item 1 is selected.

If we now reduce the value by  $\text{val}[1]$ , i.e., 50,  $V$  becomes 0. So we stop here. Thus, finally, items 3 and 1 are selected.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def knapSack(self, W, wt, val):
3.         n = len(val)
4.         K = [[0 for x in range(W + 1)] for x in range(n +
1)]
5.         # Build table K[][] in bottom up manner
6.         for i in range(1, n + 1):
7.             for w in range(W + 1):
8.
9.                 if i == 0 or w == 0:
10.                     K[i][w] = 0
11.                 elif wt[i-1] <= w:
12.                     K[i][w] = max(val[i-1] + K[i-1][w-wt[i-
1]], K[i-1][w])
13.                 else
14.                     K[i][w] = K[i-1][w]
15.
16.
17.     #Find list of selected items
18.     i=n
19.     j=W      #Bottom right corner
20.     v = K[i][j]
21.     while v:
22.         while i:
23.             if K[i][j] > K[i-1][j]:
24.                 print(i-1, ' Selected')
25.                 break
26.             else: i -= 1
```

```

27.           i -= 1
28.           j -= wt[i]
29.           v -= val[i]
30.       return K[n][W]
31. #Driver program to test above function
32. sol=Solution()
33. val = [60,50,70,30]
34. wt = [5,3,4,2]
35. W = 5 #Ans 80
36. print(sol.knapSack(W, wt, val))

```

### **Output:**

```

3 is selected Selected
1 is selected Selected
80

```

### **Complexity Analysis**

The complexity analysis is as follows:

**Time complexity:** If the number of items is  $n$ , and the knapsack capacity is  $W$ , we have to work out  $nW$  solutions. Hence, the time complexity is  $O(nW)$ .

**Space complexity:** Due to the memory requirement for table, the space complexity is  $O(nW)$ .

### **9.3 Question 65 – How to maximize the sales of a door-to-door salesman?**

Let us apply the principles of dynamic programming learned so far to a new problem.

#### **Problem statement**

You are a door-to-door salesman planning to visit the houses along a street. Each house has a certain amount of money earmarked for the purchases from salesmen like you. But you give different deals to different people and

you don't want neighbors discussing your deals across the fence. So, the constraint you should observe is that you should not visit two adjacent houses.

You have access to their income tax returns and from these, you can deduce the amount they are willing to spend. This information is stored in an array named `nums`. Determine the maximum amount of money you can collect today.

(Originally, this problem was framed as a robber problem. But I felt that it was in bad taste, so I changed it to a salesman.) Let us illustrate this with the help of two examples.

**Example 1:**

**Input:** [1,2,3,1]

**Output:** 4

**Explanation:** Sell to house 0 (money = 1), and then to house 2 (money = 3).

Total amount you can sell =  $1 + 3 = 4$ . This is the maximum.

**Example 2:**

**Input:** [2,7,9,3,1]

**Output:** 12

**Explanation:** Sell to house 0 (money = 2), sell to house 2 (money = 9), and sell to house 4 (money = 1).

Total amount you can sell =  $2 + 9 + 1 = 12$ . This is the maximum.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def sell(self, nums: List[int]) -> int:
```

## Strategy

According to the principle of dynamic programming, let us split the problem into small steps. Instead of finding a strategy for  $n$  houses directly,

we will find the optimum strategy for only one house, then only two houses, and so on. At each step, we will decide whether we should visit the house or not. Consider the second example where the `nums` array is [2,7,9,3,1] as shown in [Table 9.4](#):

i	0	1	2	3	4
nums[i]	2	7	9	3	1

**Table 9.4:** Amount collectible from each house

We will create a memoisation array such that its  $i^{\text{th}}$  element contains the maximum amount collectible up to the house  $i$ . It is customary to name such arrays as `dp` (which denotes dynamic programming). So, let us name it as `dp`.

Now, consider that you are standing in front of some house, say the fourth house, that is,  $i=3$ . How will you decide whether you should visit this house? Let us weigh the options using the data from [Table 9.4](#):

1. If you visit, then you will get an order of  $\text{nums}[3] = \text{Rs. } 3$  here but you will lose an order of  $\text{nums}[2] = \text{Rs. } 9$  from the previous house (house no. 2). The amount collectible will be  $\text{dp}[i-2] + \text{nums}[i]$ .
2. If you don't visit and skip house no. 3, then you can take whatever is the maximum amount collectible up to house no. 2, which is nothing but  $\text{dp}[i-1]$ , that is,  $\text{dp}[2]$ .

We will make the choice such that that the  $\text{dp}[i]$  is maximized. This leads us to the recurrence relation:

$$\text{dp}[i] = \max(\text{dp}[i-2] + \text{nums}[i], \text{dp}[i-1]) \quad \dots\dots(1)$$

We do not know which is better until we know the decision of the next house. So we will keep both the answers ready and proceed. We have to proceed till the last house. Here, we will decide the better option. Now, we can go backwards and decide the choice for each house.

We must assign  $\text{dp}[0]$  and  $\text{dp}[1]$  manually:

$$\text{dp}[0] = \text{nums}[0] = 2.$$

$$\text{dp}[1] = \max(\text{nums}[0], \text{nums}[1]) = \max(2, 7) = 7.$$

We can find all the other values of dp using the recurrence relation (1):

$dp[2] = \max(dp[2-2] + \text{nums}[2], dp[2-1]) = \max(2+9, 7) = 11.$   
 $dp[3] = \max(dp[3-2] + \text{nums}[3], dp[3-1]) = \max(7+3, 11) = 11.$   
 $dp[4] = \max(dp[4-2] + \text{nums}[4], dp[4-1]) = \max(11+1, 11) = 12.$

At the end, the dp array is, [2, 7, 11, 11, 12].

The answer is the last value, that is,  $dp[n-1]=12$ .

It is not asked in the question but we can find the list of houses visited as follows:

If  $dp[i] > dp[i-1]$  and house  $i+1$  is not selected, we have selected house  $i$ . Let us start from  $i = n-1 = 4$ .

$dp[4] = 12$  and  $dp[3] = 11$ . House 5 does not exist. So, we have selected house 4.

House 3 cannot be selected because house 4 is selected.

$dp[2] = 11$  and  $dp[1] = 7$ . So, we have selected house 2.

House 1 cannot be selected because house 2 is selected.

House 0 is selected because house 1 is not selected.

Thus, the selected houses are 0,2,4. The amount collected is  $2+9+1 = 12$ .

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def sell(self, nums: List[int]) -> int:
4.         n = len(nums)
5.         if(n == 0):
6.             return 0
7.         dp = [0] * n #Create an array with n zeroes
8.         dp[0] = nums[0]
9.         dp[1] = max(nums[0], nums[1])
10.        for i in range(2,n):
```

```

11.         dp[i] = max(dp[i-1], dp[i-2]+nums[i])
#Recurrence relation
12.         #Print the list of selected houses
13.         nextSelected = False
14.         for i in range(n-1,0,-1):
15.             if (nextSelected == False) and (dp[i] > dp[i-1]):
16.                 print(i, " is selected", "Sales=",nums[i])
17.                 nextSelected = True
18.             else: nextSelected = False
19.         if nextSelected == False:
20.             print(0, " is selected", "Sales=",nums[0])
21.         return dp[n-1]
22. sol = Solution()
23. print("Total sale=",sol.sell([2,7,9,3,1]))

```

### **Output:**

```

4 is the selected house, Sales= 1
2 is the selected house, Sales= 9
0 is the selected house, Sales= 2
Total sale= 12

```

## **Complexity Analysis**

The complexity analysis is as follows:

### **Time complexity:**

The loop runs  $n$  times and the run time of each iteration is independent of  $n$ . So, the time complexity is  $O(n)$ .

### **Space complexity:**

We are creating an array  $dp$  of length  $n$ . So, the space complexity is  $O(n)$ .

## **9.4 Question 66 – What is the best time to buy and sell stock?**

Let us study an application of dynamic programming into the stock market.

## **Problem statement**

Suppose you are a trader of a commodity (for example, shares) which has predictable cyclical price movements. You are given an array for which the  $i^{\text{th}}$  element is the expected price of a given stock on day  $i$ .

You are permitted to complete at the most only two transactions (that is, buy one and sell one). Write a program to find the maximum possible profit.

Note that you cannot sell a stock before you buy one. Let us illustrate this with the help of two examples.

### **Example 1:**

**Input:** [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 1 (price = 1), and sell on day 4 (price = 6). Therefore, the profit =  $6-1 = 5$ .

### **Example 2:**

**Input:** [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, the price is continuously falling, so no transaction can be done, and the maximum profit is 0.

## **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def maxProfit(self, prices: List[int]) -> int:
```

## **Strategy**

We will consider two approaches to solve this problem. The first one is the brute force approach, which is simple to implement but costly in time. The second one is based on the dynamic programming.

## Approach 1 – Brute force

We have to find the maximum difference between two numbers in the given array, which is actually the maximum profit. Also, the second number (selling price) must be larger than the first one (buying price), and it must come after buying.

In formal terms, we need to find `max(prices[j]-prices[i])` for every i and j such that  $j > i$ .

This can be easily accomplished by two nested loops.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def maxProfit(self, prices: List[int]) -> int:
4.         mprof = 0
5.         for i in range(len(prices)):
6.             for j in range(i+1, len(prices)):
7.                 mprof = max(mprof, prices[j]-prices[i])
8.         return mprof
9.
10. sol = Solution()
11. print(sol.maxProfit([7,1,5,3,6,4]))
```

**Output:** 5

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

There are two nested loops. So, the time complexity is  $O(n^2)$ .

### **Space complexity:**

We don't need extra space. So, the space complexity is  $O(1)$ .

## Approach 2 – Dynamic programming

The pattern of this problem is different from the pattern of the previous problems. Here, we will split the problem into small steps, but the recurrence relations and memoisation are not used. Let us take example 1, where prices = [7,1,5,3,6,4]. Observe the peaks and valleys in the price graph shown in [Figure 9.2](#):



**Figure 9.2:** 6 days forecast of price of a stock

We need to find the largest peak following the smallest valley. Let us do this in a single scan of prices. We will build the solution based on the decisions made every day. For this purpose, we will maintain two variables:

1. `buyPrice`, corresponding to the lowest valley found so far.
2. `profit` (maximum difference between selling price and `buyPrice`), obtained so far.

The daily decision is based on today's price as follows:

1. If today's price is lower than the current `buyPrice`, then we will update `buyPrice`.

2. Otherwise, we will calculate the profit realizable by selling the stock today and update the profit if we can improve it. In the case of continuously rising prices, we will revise the profit every day. It doesn't mean that we actually sell every day. We just want to find the profit realizable.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def maxProfit(self, prices: List[int]) -> int:  
3.         buyPrice = 1e6  
4.         profit = 0  
5.         for i in range(len(prices)):  
6.             if(prices[i] < buyPrice):  
7.                 buyPrice = prices[i]  
8.             else  
9.                 profit = max(profit, prices[i]-buyPrice)  
10.  
11.         return profit
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The loop runs only once. So, the time complexity is  $O(n)$ .

### **Space complexity:**

We don't need extra space. So, the space complexity is  $O(1)$ .

## 9.5 Question 67 – What is the best time to buy and sell stock, part 2?

This is a slight variation of the last problem. In the last problem, you were asked to make only 1 buy and 1 sell transaction. Now that restriction is gone. Now you can make any number of transactions.

### **Problem statement**

Suppose you are a trader of a commodity (for example, shares) which has predictable cyclical price movements. You are given an array for which the  $i^{\text{th}}$  element is the price of a given stock on day  $i$ .

You may buy and sell any number of times, the only restriction being that you must sell before you buy again. Design an algorithm to find the maximum profit. Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:** [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 1 (price = 1), and sell on day 2 (price = 5). Profit =  $5-1 = 4$ .

Then, buy on day 3 (price = 3), and sell on day 4 (price = 6). Profit =  $6-3 = 3$ .

Total profit =  $4+3 = 7$ .

#### **Example 2:**

**Input:** [3, 7, 2, 3, 6, 7, 6, 7].

**Output:** 10

**Explanation:** Buy on day 0 (price = 3), and sell on day 1 (price = 3). Profit =  $7-3 = 4$ .

Then, buy on day 2 (price = 2), and sell on day 5 (price = 7). Profit =  $7-2 = 5$ .

Then, buy on day 6 (price = 6), and sell on day 7 (price = 7). Profit =  $7-6 = 1$ .

Total profit =  $4+5+1 = 10$ .

### Example 3:

**Input:** [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 0 (price = 1), and sell on day 4 (price = 5). Profit =  $5-1 = 4$ .

### Example 4:

**Input:** [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, the price is continuously falling, so there is no scope for booking profit. So, the profit will be 0.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def maxProfit(self, prices: List[int]) -> int:
```

## Strategy

According to the principles of dynamic programming, we will take a daily trading decision. As there is no restriction on the number of transactions, you may buy on every valley and sell on every peak. But, how will you take care of the requirement that you cannot have two consecutive buy or sell deals. The trick is to use the fact that you just have to report the total profit. You don't have to actually sell or buy every day. So, we will simply book notional profit every day. For days in the range  $1 \dots n$ , the decision rule will be as follows:

- If today's price is more than yesterday's price, we will sell.
- If today's price is less than yesterday's price, we will buy.

These rules can be implemented with the statement:

```
if prices[i] > prices[i-1]:
```

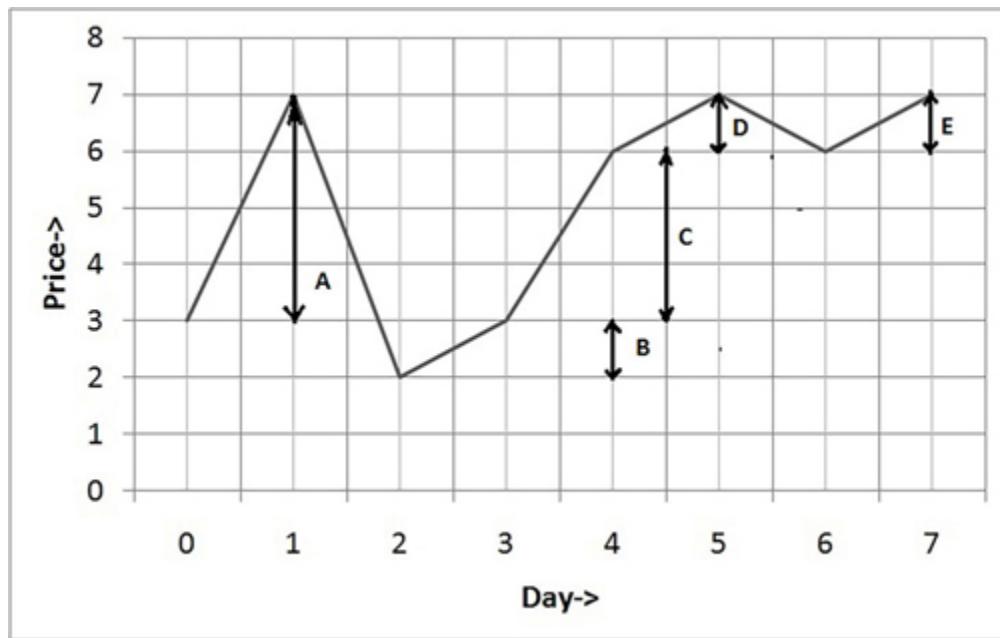
```
ans+=(prices[i] - prices[i-1])
```

Let us see if these simple rules are able to take care of all the corner cases.

In example 1, prices = [7,1,5,3,6,4]. Here, we buy on every dip and sell on every rise. The last dip(from 6 to 4) is ignored because we book profit only when we complete a buy-sell cycle. There is no scope for completing it. So profit is:

$$(5-1) + (6-3) = 7$$

Consider *Example 2*. The prices = [3, 7, 2, 3, 6, 7, 6, 7]. The price chart is shown in [Figure 9.3](#):



*Figure 9.3: Profit scenarios*

Between day 0 and day 1 there is a rise of 4 (shown by arrow A). On days 3,4, and 5, there is a continuous rise in the price. We will add up the notional profits shown by the arrows B, C, and D, amounting to  $1+3+1 = 5$ . This is equivalent to buying on day 2 and selling on day 5. The last profit is booked on day 7 as shown by the arrow E. The total profit is 10.

In example 3, prices = [1,2,3,4,5]. Here as well, we will book the notional profit every day. It is equivalent to buying on the first day and selling on the last day.

In example 4, prices = [7,6,4,3,1]. The price is a free fall (great depression). Here, there is no scope of making a profit.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def maxProfit(self, prices: List[int]) -> int:  
3.         ans = 0  
4.         for i in range(1, len(prices)):  
5.             if prices[i] > prices[i-1]:  
6.                 ans+=(prices[i] - prices[i-1])  
7.         return ans  
8. sol = Solution()  
9. print(sol.maxProfit([3, 7, 2, 3, 6, 7, 6, 7]))
```

**Output:** 10

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The loop runs  $n$  times and the run time of each iteration is independent of  $n$ . So, the time complexity is  $O(n)$ .

### **Space complexity:**

We don't need extra space. So, the space complexity is  $O(1)$ .

## 9.6 Question 68 – In how many ways can you climb the stairs?

This is a classic example of the dynamic programming that requires recurrence relations and memoisation.

## Problem statement

You are climbing a staircase having  $n$  steps. But you are energetic and sometimes you can take a double step. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? Note that  $n$  will be a positive integer. Let us illustrate this with the help of some examples.

**Example 1:**

**Input:** 2

**Output:** 2

**Explanation:** There are two ways to climb to the top if there are two steps to climb:

1. single + single
2. double

**Example 2:**

**Input:** 3

**Output:** 3

**Explanation:** There are three ways to climb to the top if there are three steps to climb.

1. single + single + single
2. single+ double
3. double+ single

**Solution format**

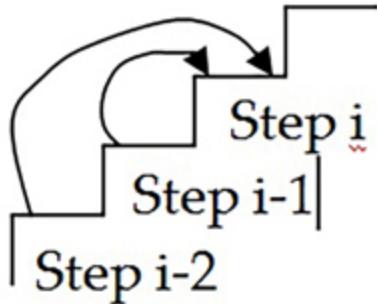
Your solution should be in the following format:

1. class Solution:
2.     def climbStairs(self, n: int) -> int:

**Strategy**

Let us use the dynamic programming. We will not directly think of climbing  $n$  steps. We will work out a bottoms-up approach. We will examine how we can climb 1 step, then two steps, then three steps, and so

on till we reach  $n$  steps. Then we will find a recurrence relation that will enable us to find the answer for the  $i^{\text{th}}$  step in terms of the previous answers:



**Figure 9.4:** Two ways of reaching step  $i$

Let us think of the problem at the  $i^{\text{th}}$  step. How can you reach step  $i$ ? As shown in [Figure 9.4](#), either you take a single step from step  $i-1$  or take a double step from step  $i-2$ . If there are  $F_{i-1}$  ways to reach step  $i-1$ , and if there are  $F_{i-2}$  ways to reach step  $i-2$ , the number of ways in which step  $i$  can be reached is as follows:

$$F_i = F_{i-1} + F_{i-2}$$

Let us tabulate the results for values of  $n$  in the range 0...5 in [Table 9.5](#):

N	Patterns	No. of patterns(Fn)
0	0	1
1	1	1
2	2, 11	2
3	12, 21, 111	3
4	22, 112, 121, 211, 1111	5

**Table 9.5:** Number of patterns for climbing  $n$  steps.

Does it ring a bell? This is similar to the Fibonacci sequence. We have already discussed it in detail in [Section 9.1](#). The iterative code which gives the time complexity of  $O(n)$  is:

```

1. class Solution:
2.     def fib(self, n: int) -> int:
3.         f_n1 = 1

```

```

4.         f_n2 = 0
5.         if n <= 1:
6.             return n
7.         else
8.             for i in range(n-1):
9.                 f = f_n1 + f_n2
10.                f_n2 = f_n1
11.                f_n1 = f
12.            return f
13. #Driver code.
14. sol = Solution()
15. for j in range(6):
16.     print(j,sol.fib(j))

```

The output is summarized in [Table 9.6](#):

<b>n</b>	<b>Fibonacci(n)</b>
0	0
1	1
2	1
3	2
4	3
5	5

*Table 9.6: Fibonacci numbers*

If we compare [Table 9.2](#) and [Table 9.3](#), we can conclude that the number ways to climb  $n$  stairs is  $\text{Fibonacci}(n+1)$ .

Now, we will study the various approaches to solve the problem.

## Approach 1 – Pure dynamic programming

A pure dynamic programming approach using the bottoms-up iterative methodology is as follows:

We will create a memoisation array  $dp$  having  $n+1$  zero elements. We will initialize  $dp[1]$  to 1 and  $dp[2]$  to 2. Then, we will run a loop where the

loop variable  $i$  goes from 3 to  $n$ . In this loop, we will use the recurrence relation to fill the array.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def climbStairs(self, n: int) -> int:
3.         if (n == 1):
4.             return 1;
5.         dp = [0] * (n+1) #Create an array of n+1 zeroes.
6.         dp[1] = 1
7.         dp[2] = 2
8.         for i in range(3,n+1):
9.             dp[i] = dp[i - 1] + dp[i - 2]
10.            return dp[n]
```

This code gives same output as [Table 9.5](#).

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

As the loop runs only once, the time complexity is  $O(n)$ . However, due to the iterative method, the function calling overhead of recursive method is avoided.

### **Space complexity:**

The stack requirement of the function calling is avoided but we still require a memoisation array of length  $n$ , so space complexity is still  $O(n)$ .

## Approach 2 – Compact array

If you see the requirement of previous results carefully, you will observe that for filling  $dp[i]$  you need only two previous values, namely  $dp[i-1]$  and  $dp[i-2]$ . There is no need to maintain a full array of length  $n$ . We can modify the loop as follows:

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def climbStairs(self, n: int)-> int:
3.         if n==0:
4.             return 1
5.         if n <= 3:
6.             return n
7.         f_n1 = 2
8.         f_n2 = 1 \
9.         for i in range(3,n+1):
10.             f = f_n1 + f_n2
11.             f_n2 = f_n1
12.             f_n1 = f
13.         return f
14. #Driver code
15. sol = Solution()
16. for j in range(6):
17.     print(j,sol.climbStairs(j))
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

As the loop runs only once, the time complexity is  $O(n)$ .

### **Space complexity:**

As we don't require an array, the space complexity is  $O(1)$ .

We will now introduce two more methods that achieve a still better time complexity.

But before that, let us study an interesting problem solved by an Indian poet and mathematician, Acharya Hemchandra. He was a scholar of poetic meters. The syllables of lyrics can either be short or long. The long syllable (for example, aa, ee, oo) take twice as much time to pronounce as the short

ones (for example, u, i). In Sanskrit, they are called the *Laghu matra* and the *Guru matra*. For a music composition, the composer wants the time duration of every line to be same. Suppose, the rhythm cycle consists of  $n$  beats. Acharya Hemchandra wanted to find out how many patterns of Laghu(1 beat) and Guru(2 beats) are possible to make total of  $n$  beats. Such patterns are called cadences. He provided an algorithm to enumerate all the possible cadences of length  $n$ . According to his algorithm, you can create an  $n$  beat cadence by appending a Guru (G) to each cadence of length  $n-2$  or by appending a Laghu(L) to each cadence of length  $n-1$ . This is illustrated in [Table 9.7](#), starting from  $n=0$ :

N	Patterns	No. of patterns
0	$P(0) = \text{None}$	1
1	$P(1) = L$	1
2	$P(2) = P(0) + G, P(1) + L = G, LL$	2
3	$P(3) = P(1) + G, P(2) + L = LG, GL, LLL$	3
4	$P(4) = P(2) + G, P(3) + L = GG, LLG, LGL, GLL, LLLL$	5

**Table 9.7:** Constructing  $P(i)$  from  $P(i-1)$  and  $P(i-2)$

We can conclude that the number of cadence patterns for an  $n$  beat rhythm is  $\text{Fibonacci}(n+1)$ . We should rather call it  $\text{Hemchandra}(n+1)$  because Acharya Hemchandra published this remarkable result in the year 1100, that is, 50 years before Fibonacci.

Mathematicians have been fascinated by the Fibonacci numbers because they occur in many natural phenomena. Dan Brown, author of the famous novel, Da Vinci Code, has devoted 4 pages on Fibonacci numbers in the novel. Let us now study the algorithms that give a better time complexity. This is a bit diversion from the dynamic programming but it is worth it.

Let us study the following approach.

### Approach 3 – Matrix exponentiation

Now let us return to the subject of improving the time complexity. What is better than  $O(n)$ ? It is  $O(\log n)$ .

Consider the matrix:

$$\text{B} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

It has been observed that:

$$\text{B}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

## Python code

To verify the above rule, let us write a small Python program:

```
1. import numpy as np  
2. b=np.array([[1,1],[1,0]])  
3. print(b@b@b@b) #Note that @ operator in numpy does matrix multiplication.
```

## **Output:**

```
[[5 3]  
 [3 2]]
```

The output is matrix  $b^4$ . Verify that  $b^4[0][1] = b^4[1][0] = \text{Fibo}(4) = 3$ . So for finding  $\text{Fibo}(n)$ , we need to raise the matrix  $b$  to the power  $n$ . Instead of multiplying the matrix  $n$  times, we can find a better solution.

Let us express  $n$  in a binary search.

For example, let  $n = 13 = 1101_2$ .

$$n = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = 8 + 4 + 0 + 1$$

If the matrix  $b$  has to be raised to the power 13, we can write:

$$b^{13} = b^{2^3 * 1} + b^{2^2 * 1} + b^{2^1 * 0} + b^{2^0 * 1}$$

$$b^{13} = b^8 * b^4 * b$$

We will start with  $b$  and multiply it with  $b^{2^i}$ , if the  $i^{\text{th}}$  bit in the binary expansion of  $n$  is 1. Note that  $(b^{2^i})^2 = b^{(2^i)*2} = b^{2^{i+1}}$ . Thus, we get the multiplier for the next step by squaring the current multiplier.

Let us define a matrix  $a$  as the multiplier. It will initially be equal to  $b$  but gets squared in each iteration. We will initialize the result with the identity matrix. If the LSB in the binary representation of  $n$  is 1, we multiply the result with  $a$ . Then, we shift  $n$  right and square  $a$ .

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def climbStairs(self, n:int) ->int:
3.         n=n+1 #We need Fibo(n+1)
4.         a = np.array([[1,1],[1,0]]) #Powers of b
5.         b = np.array([[1,1],[1,0]]) #Binet matrix
6.         result = np.identity(2, dtype = int) # Initially
result is identity matrix
7.         while n>0:
8.             if n & 1: #Check LSB of n
9.                 result = result @ a #If LSB(n)== 1 then
multiply result by a
10.            n >= 1 # shift n right
11.            a = a @ a # Square a
12.        return result[0][1]
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

The number of bits required to express  $n$  is  $\log n$ . So, the time complexity is  $O(\log n)$ .

### **Space complexity:**

We don't need extra space. So, the space complexity is  $O(1)$ .

Now let us study the most efficient approach.

## Approach 4 – Binet's formula

We have achieved a time complexity of  $O(\log n)$  with approach 3. Can there be a way of achieving  $O(1)$ ? Yes, there is. We can use the Binet formula. We can derive it by solving a different equation:

$$y[n] = y[n-1] + y[n-2]$$

with initial conditions:  $y[0] = 0$  and  $y[1] = 1$

The solution is:

$$y[n] = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}$$

We will need to convert the result to an integer because the above formula uses floating point calculations, and it may not result in exact integers.

## Python code

The following code implements this strategy:

```
1. class Solution:
2.     def climbStairs(self, n:int) ->int:
3.         n=n+1    # In climb stairs problem, we need Fibo(n+1)
4.         sqrt5 = math.sqrt(5)
5.         return int(((1 + sqrt5) ** n - (1 - sqrt5) ** n) /
(2 ** n * sqrt5)))
6. sol = Solution()
7. for j in range(6):
8. print(j,sol.climbStairs(j))
```

## **Output:**

```
0 1
1 1
2 2
3 3
4 5
5 8
```

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

There is no loop. So, the time complexity is  $O(1)$ .

### **Space complexity:**

We don't need extra space. So, the space complexity is  $O(1)$ .

## 9.7 Question 69 – How will you minimize the cost of climbing stairs?

This is similar to the last question but now the staircase is no longer free. Each step is associated with a cost. If you take that step, you have to pay that cost. Find a sequence that minimizes the cost. Again, this is a classic application of the dynamic programming.

### Problem statement

You are given an array named `cost`, containing the cost of each step. It is indexed 0.

You need to find a sequence of single or double steps to get the minimum cost to climb all steps. You can either start from the step with the index 0, or the step with the index 1. Let us illustrate with the help of two examples.

#### **Example 1:**

**Input:** `cost = [10, 15, 20]`

**Output:** 15

**Explanation:** The sequence 0, 2 costs  $10 + 20 = 30$ . The cheapest is to start on step 1, pay 15 and go to the top by taking a double step.

#### **Example 2:**

**Input:** `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

**Output:** 6

**Explanation:** To avoid 100 on step 1, use step 0 and take a double step to reach step 2. Then, to avoid 100 on step 5, skip step 3, take step 4 and skip 5. Then, take steps 6, 7, and 9.

So, the cheapest sequence is 0,2,4,6,7,9. The cost is 6.

**Note:**

The number of stairs will be in the range [2, 1000].

Every  $\text{cost}[i]$  will be an integer in the range [0, 999].

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def minCostClimbingStairs(self, cost: List[int]) -> int:
```

## Strategy

We will use the dynamic programming principle. At every step  $i$ , we have to decide: should we take this step or skip it? We cannot finalize the decision unless we know the decision of the next step. So we will consider both the options and store the results in the memorization array  $\text{dp}$ . It will be a  $n \times 2$  array.  $\text{dp}[i][0]$  represents the cost up to the step  $i$  if we take step  $i$ .  $\text{dp}[i][1]$  represents the cost up to the step  $i$  if we skip it. Initially, it will all be zeros. At each step, we will fill the  $i^{\text{th}}$  row of the  $\text{dp}$ . So, we will build the  $\text{dp}$  in a bottoms-up manner. When we reach the end, we can take a decision. Then, we will propagate it backwards to all the steps.

If we take step 0, the cost will be  $\text{cost}[0]$  and if we skip step 0, the cost will be 0. Therefore, we will initialize  $\text{dp}$  as:

$\text{dp}[0][0] = \text{cost}[0]$  and  $\text{dp}[0][1] = 0$ .

Now we will find a recurrence relation that enables us to compute the  $i^{\text{th}}$  row in terms of the previous rows.

The cost at the step  $i$  depends on the decision of skipping it.

- **Not taking the current step:** We must take the previous step so that we can skip this one. The cost will be just the cost of taking step  $i-1$ .

So  $\text{dp}[i][1] = \text{dp}[i-1][0]$

- **Taking the current step:** Add  $\text{cost}[i]$  to the best decision of step  $i-1$ .

So  $dp[i][0] = \min(dp[i-1][0], dp[i-1][1]) + cost[i]$

To clarify, we will walk through the algorithm with the cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1].

The length of the cost array is 10. The variable  $i$  goes from 1 to 9.

Initially:

$dp[0][0] = 1$

$dp[0][1] = 0$

## Consider i = 1

If we take the current step, we will add the cost of the current step to the best decision of the previous step. The best decision is  $dp[0][1] = 0$ :

So  $dp[1][0] = 0 + cost[0] = 0+100 = 100$

If we don't take the current step, we will carry forward the cost of taking the previous step:

So  $dp[1][1] = dp[0][0] = 1$

The first two rows of the dp array are shown in [Table 9.8](#):

I	Take step	Don't take step
0	1	0
1	100	1

**Table 9.8:** dp after the first step

## Consider i = 2

If we take the current step, we will add the cost of the current step to the best decision of the previous step. The best decision is  $dp[1][1] = 1$ :

So  $dp[2][0] = 1 + cost[2] = 1+1 = 2$

If we don't take the current step, we will carry forward the cost of taking the previous step:

So  $dp[2][1] = dp[1][0] = 100$

The first three rows of the dp array are shown in [Table 9.9](#):

I	Take step	Don't take step
0	1	0
1	100	1
2	2	100

**Table 9.9:** dp after the second step

Contents of the dp at the end of the loop are shown in [Table 9.10](#):

I	Take step	Don't take step
0	1	0
1	100	1
2	2	100
3	3	2
4	3	3
5	103	3
6	4	103
7	5	4
8	104	5
9	6	104

**Table 9.10:** dp after 9 steps

Now, we are in a position to take a decision. The choice is between 6 and 104. Obviously, we will choose 6, that is, we will take step 9. Thus, the final answer is 6.

It is not required in the problem, but if we want to trace the sequence of the steps, we need to proceed backward from the final decision. Just remember that a step  $i-1$  cannot be skipped if the step  $i$  is skipped.

Step 9 is taken because  $6 < 104$

Step 8 is skipped because  $104 > 5$

Step 7 cannot be skipped even though  $5 > 4$ , because step 8 is skipped.

Step 6 is taken because  $4 < 103$

Step 5 is skipped because  $103 > 3$

Step 4 cannot be skipped even though  $3 = 3$ , because step 5 is skipped.

Step 3 is skipped because  $3 > 2$

Step 2 is taken because  $2 < 100$

Step 1 is skipped because  $100 > 1$

Step 0 cannot be skipped even though  $1 > 0$ , because step 1 is skipped.

Thus, the sequence of the steps is 0,2,4,6,7,9. You may like to write a Python code for doing this. Take hints from question 65, i.e., the door-to-door salesman.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. import numpy as np
3. class Solution:
4.     def minCostClimbingStairs(self, cost: List[int]) -> int:
5.         n=len(cost)
6.         dp = np.zeros((n+1,2))
7.
8.         for i in range(1, len(cost)+1):
9.             dp[i][0] = min(dp[i-1][0], dp[i-1][1])+ cost[i-1]#Taking curr step
10.            dp[i][1] = dp[i-1][0] # not taking current step
11.
12.        return int(min(dp[n][0], dp[n][1]))
13. sol = Solution()
14. print(sol.minCostClimbingStairs([1, 100, 1, 1, 1, 100, 1, 1, 100, 1]))
```

**Output:** 6

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:**

As the loop runs once, the time complexity is  $O(n)$ .

**Space complexity:**

Due to the memory required for the dp, the space complexity is  $O(n)$ .

## 9.8 Question 70 – What is the minimum number of coins to dispense the change?

This is another application of the dynamic programming.

### Problem statement

You have a large number of coins of different denominations, say Rupees. You need to pay a given amount of money using the minimum number of coins. Write a function to do that and return the number. If that amount of money cannot be made up by any combination of the coins, return -1. Let us illustrate with the help of two examples.

**Example 1:**

**Input:** coins = [1, 2, 5], amount = 11 Rs

**Output:** 3

**Explanation:** Use 2 coins of Rs. 5, and one coin of Re. 1.  $11 = 5 + 5 + 1$

**Example 2:**

**Input:** coins = [2], amount = 3 Rs

**Output:** -1

**Explanation:** You cannot pay 3 Rs with the coins of 2 Rs alone.

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def coinChange(self, coins: List[int], amount: int) ->  
int:
```

## Strategy

Using the principle of dynamic programming, we will find the solution incrementally. We will find the minimum number of coins for paying Rs. 0, then Rs. 1, and so on. We will create an array  $dp$  with  $(amount+1)$  elements. The  $i^{\text{th}}$  element of this array will contain the minimum number of coins needed to make a payment of Rs  $i$ . The array will be initialized with a large number.  $dp[0]$  is made 0 because for paying 0, you need 0 coins. We will start filling the array from 0 upwards till we reach the specified amount.

$\text{coins}[0]$ ,  $\text{coins}[1]$ ,  $\text{coins}[2]$  are the denominations of the coins available. Let us illustrate the recurrence relation with the help of the first example. Here,  $\text{coins}[0]=1\text{Re}$ ,  $\text{coins}[1]=2\text{Rs}$ ,  $\text{coins}[2]=5\text{Rs}$ .

At step  $i$ , we have three choices to consider:

1. Add a 1 Re coin to the collection  $i-1$
2. Add a 2 Rs coin to the collection  $i-2$
3. Add a 5 Rs coin to the collection  $i-5$

We will choose the minimum of the three options to get  $dp[i]$ .

Initially,  $dp = [0, 10000, 10000, 10000 \dots]$

### Consider $i = 1$

We can add 1 Re coin to the nil amount. We cannot use coins of 2 Rs and 5 Rs because we don't have  $dp[1-2]$  and  $dp[1-5]$  present. So, the minimum coin count = 1:

$dp = [0, 1, 10000, 10000 \dots]$

### Consider $i = 2$

We can add 1 Re coin to  $dp[1]$  (count =  $1+1=2$ ) or add 2 Rs coin to  $dp[0]$  (count =  $0+1=1$ ). We cannot use coins of 5 Rs because we don't have  $dp[2-5]$ . The minimum of the two choices gives a coin count of 1. So the  $dp$  is:

$dp = [0, 1, 1, 10000, 10000 \dots]$

## Consider i = 3

We can add 1 Re coin to  $\text{dp}[2]$  (count =  $1+1=2$ ), or add 2 Rs coin to  $\text{dp}[1]$  (count =  $1+1=2$ ). We cannot use coins of 5 Rs, because we don't have  $\text{dp}[3-5]$ . Both the choices give a coin count of 2. So the dp is:

$\text{dp} = [0, 1, 1, 2, 10000, 10000 \dots]$

## Consider i = 4

We can add 1 Re coin to  $\text{dp}[3]$  (count =  $2+1=3$ ), or add 2 Rs coin to  $\text{dp}[2]$  (count =  $1+1=2$ ). We cannot use coins of 5 Rs because we don't have  $\text{dp}[4-5]$ . The minimum of the two choices gives a coin count of 2. So the dp is:

$\text{dp} = [0, 1, 1, 2, 2, 10000 \dots]$

## Consider i = 5

We can add 1 Re coin to  $\text{dp}[4]$  (count =  $2+1=3$ ), or add 2 Rs coin to  $\text{dp}[3]$  (count =  $2+1=3$ ), or add 5 Rs coin to  $\text{dp}[0]$  (count =  $0+1=1$ ). The minimum of the three choices gives a coin count of 1. So the dp is:

$\text{dp} = [0, 1, 1, 2, 2, 1, 10000, 10000\dots]$

Now we can write the general recurrence relation.

$\text{dp}[i] = \min(\text{dp}[i - \text{coin}] + 1)$  for all coins where  $i > \text{coin}$ .

Continuing this way, when we reach  $i = 11$ , we get:

$\text{dp} = [0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 2, 3]$

The final answer is  $\text{dp}[11] = 3$ . Verify that it uses two coins of 5 Rs and one coin of 1 Re.

## Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def coinChange(self, coins: List[int], amount: int) ->  
int:
```

```

3.         dp = [10000]*(amount + 1)
4.         dp[0] = 0
5.         for i in range(1,amount+1):
6.             for j in coins:
7.                 if i - j < 0:continue
8.                 dp[i] = min(dp[i], dp[i-j]+1)
9.         return dp[amount] if dp[amount] != 10000 else -1
10. #Driver code
11. sol=Solution()
12. print(sol.coinChange([1, 2, 5],11))

```

### **Output:** 3

Although the above method is more intuitive, we can change the order of the loops and save one comparison. This method is more suitable for the next problem.

### **Python code**

The following code implements this strategy:

```

1. class Solution:
2.     def coinChange(self, coins: List[int], amount: int) ->
int:
3.         dp = [10000]*(amount + 1)
4.         dp[0] = 0
5.         for j in coins:
6.             for i in range(j,amount+1):
7.                 dp[i] = min(dp[i], dp[i-j]+1)
8.         return dp[amount] if dp[amount] != 10000 else -1

```

### **Complexity Analysis**

The complexity analysis for both the methods is as follows:

#### **Time complexity:**

There are two nested loops. If  $n = \text{amount}$  and  $c = \text{number of coins}$ , the time complexity is  $O(nc)$ . But if  $c$  is constant, the time complexity is  $O(c)$ .

### **Space complexity:**

As we are using an array of length  $n+1$ , the space complexity is  $O(n)$ .

## **9.9 Question 71 – How many coin patterns to dispense the change?**

This is a variation of the last question. Instead of finding the minimum number of coins, find how many patterns of coins are possible.

### **Problem statement**

You have a large number of coins of different denominations. You need to pay a given amount of money using various combinations of coins. Find how many patterns of coins are possible totaling to the given amount. If that amount of money cannot be made up by any combination of the coins, return 0. Let us illustrate this with the help of some examples.

#### **Example 1:**

**Input:** amount = 5, coins = [1, 2, 5]

**Output:** 4

**Explanation:** The amount can be paid with following patterns:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

#### **Example 2:**

**Input:** amount = 3, coins = [2]

**Output:** 0

**Explanation:** With coins of 2 Rs only, you cannot pay the amount of 3 Rs.

#### **Example 3:**

**Input:** amount = 10, coins = [10]

**Output:** 1

**Explanation:** With coins of 10 Rs only, there is only 1 way of paying Rs 10.

## Solution format

Your solution should be in the following format:

1. class Solution:
2.     def change(self, amount: int, coins: List[int]) -> int:

## Strategy

We will find the incremental solutions in steps of 1 Re. Starting from the amount of 0 Rs, we will work up to the final amount. We will first create an array  $dp$  with  $(amount+1)$  elements, all zeroes.  $dp[j]$  represents the number of patterns for amount  $j$ . We will make  $dp[0]=1$  because when amount is 0, we don't need any coins, that is, we have 1 pattern - no coins. We will run two nested loops. The outer loop with the coin values  $i$ , and the inner loop with the amount  $j$ . Let us illustrate the algorithm as we walk through example 1.

Consider amount = 5 and coins = [1,2,5]

The initial state of the  $dp$  is shown in [Table 9.11](#):

J	0	1	2	3	4	5
dp[j]	1	0	0	0	0	0

*Table 9.11: Initial state of dp*

The outer loop defines the repertory of the coins available. Note that the loop variable  $i$  is not the index but the coin value. In the first iteration of the loop, we will assume only 1 Re coin is available. In the second iteration, coins of 1 and 2 are available. In the third iteration, all the coins are available. The idea is that when we add a new coin of value  $i$  to our repertory, and the amount to be given is  $j$ , we have an additional option of using that coin at the place  $dp[j-i]$ . So, we will add  $dp[j-i]$  to  $dp[j]$ .

## Consider i= first coin and value = 1

The inner loop runs from  $j = 1$  to 5. At every value of  $j$ , we will add  $dp[j-1]$  to  $dp[j]$ . The calculations will proceed as follows:

$$dp[1] = dp[1] + dp[0] = 1$$

$$dp[2] = dp[2] + dp[1] = 1$$

$$dp[3] = dp[3] + dp[2] = 1$$

and so on. At the end, the dp looks as shown in [Table 9.12](#):

j	0	1	2	3	4	5
dp[j]	1	1	1	1	1	1

**Table 9.12:** State of dp after the 1st step

It also means that when only 1 Re coin is available, the only way of paying amount  $j$  is to give  $j$  coins of 1 Re each.

## Now consider i=2

This means a 2 Rs coin is also available in our repertory. When we are considering the amount  $j$ , we have an option of giving this coin for amount  $j-2$ . So, we will add  $dp[j-2]$  to  $dp[j]$ . We will start the inner loop with  $j=2$  because  $dp[1-2]$  is out of bounds.

The calculations will proceed as follows:

$$dp[2] = dp[2] + dp[0] = 2$$

$$dp[3] = dp[3] + dp[1] = 2$$

$$dp[4] = dp[4] + dp[2] = 3 \quad (\text{Take new value of } dp[2])$$

and so on. At the end, the dp looks as shown in [Table 9.13](#):

J	0	1	2	3	4	5
dp[j]	1	1	2	2	3	3

**Table 9.13:** State of dp after the 2nd step

Let us interpret this table. In the previous step, there was only one way of paying Rs 3, that is, 3 coins of Re 1. So  $dp[3]$  was 1. Now, we have an option of giving Rs 2 coin. So we will add the options available for amount 1 to options available for amount 3.

## Now consider i=5

So, Rs 5 coins have been added to Rs 1 and 2 coins. We will start the inner loop with j=5 to avoid out of bounds problem. There is only one calculation in this iteration.

$$dp[5] = dp[5] + dp[0] = 3+1 = 4$$

At the end, the dp looks as shown in [Table 9.14](#):

j	0	1	2	3	4	5
dp[j]	1	1	2	2	3	4

**Table 9.14:** State of dp after the 3rd step

The final answer is  $dp[5] = 4$ .

Now let us walk through Example 2. Consider amount = 3, and coins =[2]

The initial state of dp is shown in [Table 9.15](#):

J	0	1	2	3	
dp[j]	1	0	0	0	

**Table 9.15:** Initial state of dp

## Consider i = 2 (Only one value)

j starts from 2 so that  $j - 2$  is not negative. Using the recurrence relation:

$$dp[j] = dp[j] + dp[j-2]$$

We get:

$$dp[2] = dp[2] + dp[0] = 1$$

$$dp[3] = dp[3] + dp[1] = 0$$

Now the dp looks as shown in [Table 9.16](#):

J	0	1	2	3	
dp[j]	1	0	1	0	

**Table 9.16:** State of dp after 1<sup>st</sup> step

The answer is 0 because the amount of Rs 3 cannot be made with coins of Rs 2.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def change(self, amount: int, coins: List[int]) -> int:
4.         dp = [0]*(amount + 1)
5.         dp[0] = 1
6.         for i in coins:
7.             for j in range(i, amount + 1):
8.
9.                 dp[j] += dp[j - i]
10.            return dp[amount]
11. sol=Solution()
12. print(sol.change(5,[1, 2, 5]))
13. print(sol.change(3,[2]))
14. print(sol.change(10,[10]))
```

**Output:** 4, 0, 1

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

There are two nested loops. If  $n = \text{amount}$  and  $c = \text{number of coins}$ , the time complexity is  $O(nc)$ . But if c is constant, the time complexity is  $O(n)$ .

### **Space complexity:**

As we are using an array of length  $n+1$ , the space complexity is  $O(n)$ .

## 9.10 Question 72 – How many unique paths exist in a square grid?

In this question, we have to imagine a 2-dimensional problem scenario and apply the principles of dynamic programming.

## **Problem statement**

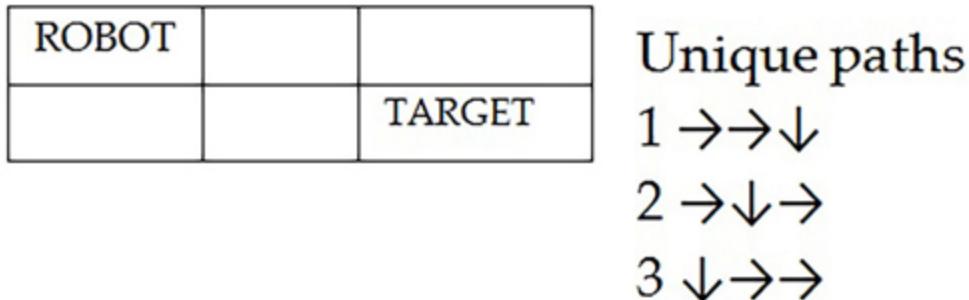
A robot is standing on the top left corner of an  $n \times m$  grid ( $n$  rows and  $m$  columns). The robot can only move either down or right at any point in time. The robot aims to reach the bottom-right corner of the grid. How many unique paths are possible for this move? Let us illustrate this with the help of two examples.

### **Example 1:**

**Input:**  $m=3$ ,  $n=2$

**Output:** 3

**Explanation:** See the grid shown in [Figure 9.5](#). There are three unique paths as shown by the arrows. They are: (1)Right, Right, Down; (2)Right, Down, Right; (3)Down, Right, Right:



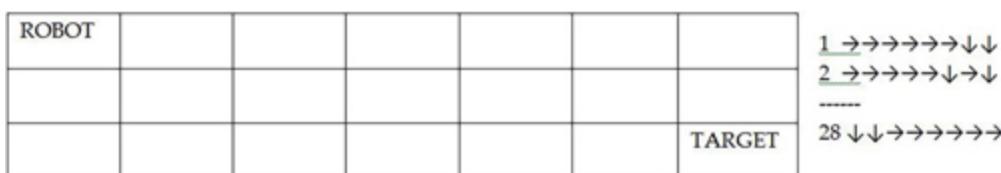
*Figure 9.5: Paths for example 1*

### **Example 2:**

**Input:**  $m=7$ ,  $n=3$

**Output:** 28

**Explanation:** See the grid shown in [Figure 9.6](#). The paths are shown by the arrows. There are 28 unique paths:



**Figure 9.6:** Paths for example 2

## Solution format

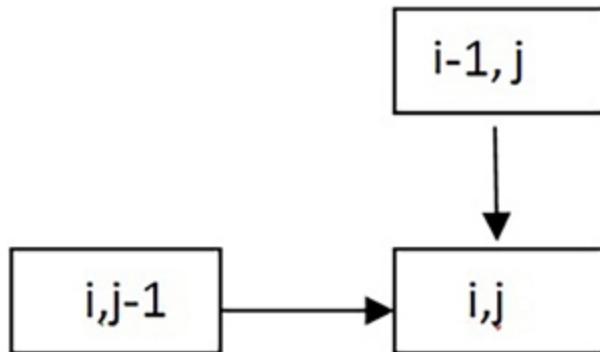
Your solution should be in the following format:

```
1. class Solution:  
2.     def uniquePaths(self, m: int, n: int) -> int:
```

## Strategy

We will apply the dynamic programming philosophy. In the previous problems, the target was linear. In this problem, it is 2-dimensional. But the methodology will be the same. Our final target is  $(m-1, n-1)$ . We will build the incremental solutions starting from  $(0,0)$ , and increasing by 1 step at a time in the  $x$  and  $y$  directions. We will create the recurrence relations which help us get the number of ways of reaching the cell  $(i, j)$  in terms of previous solutions. We will store all the intermediate results in an  $n \times m$  memoisation array, named  $dp$ . The content of the element  $dp[i, j]$  represents the number of ways of reaching the cell  $[i, j]$  from the cell  $[0, 0]$ .

For building the recurrence relation, take a look at the incoming paths into the cell  $(i, j)$  as shown in [Figure 9.7](#):



**Figure 9.7:** Incoming paths into cell  $i, j$

As we can move only in the right or down direction, the incoming paths into this cell are from the top( $i-1, j$ ), and from the left( $i, j-1$ ).

If there are  $n_1$  ways of reaching the top and  $n_2$  ways of reaching the left cell, then the number of ways for reaching the cell  $(i, j)$  are  $n_1+n_2$ . So, the recurrence relation is:

$$dp[i, j] = dp[i-1, j] + dp[i, j-1] \quad \dots(1)$$

Let us now create the initial conditions. The dp array should initially be all zeroes. Now, consider the top row and the left column. Let us take the top row first. As the robot moves only right and down, the only way you can reach any cell in this row is by a series of right movements. Thus, there is only one way to reach any cell in the top row. So we will fill it with 1s. Likewise, there is only one way to reach any cell in the left column. So we will fill the left column also with 1s. For illustration, let us consider  $m=n=3$ . The dp array after the initialization looks as shown in [Table 9.17](#):

1	1		1
1			
1			TARGET

**Table 9.17:** dp after initialization

Let us apply the recurrence relation (1) to the second row:

$$\begin{aligned} dp[i, j] &= dp[i-1, j] + dp[i, j-1] \\ dp[1, 1] &= dp[0, 1] + dp[1, 0] = 1 + 1 = 2 \\ dp[1, 2] &= dp[0, 2] + dp[1, 1] = 1 + 2 = 3 \quad (\text{Take new value of } \\ &\quad dp[1, 1]) \end{aligned}$$

The resulting state of the dp is shown in [Table 9.18](#):

1	1	1
1	2	3
1		

**Table 9.18:** dp after second row processing

Now let us apply the recurrence relation to the third row:

$$\begin{aligned} dp[2, 1] &= dp[1, 1] + dp[2, 0] = 2 + 1 = 3 \\ dp[2, 2] &= dp[1, 2] + dp[2, 1] = 3 + 3 = 6 \quad (\text{Take new value of } \\ &\quad dp[2, 1]) \end{aligned}$$

The resulting state of the dp is shown in [Table 9.19](#):

1	1	1
---	---	---

1	2	3
1	3	6

*Table 9.19: dp after third row processing*

The contents of the bottom right corner are 6, so the answer is 6.

## Python code

The following code implements this strategy:

```

1. class Solution:
2.     def uniquePaths(self, m: int, n: int) -> int:
3.         #Create n X m array of zeroes
4.         dp = [[0 for x in range(m)] for y in range(n)]
5.         for i in range(m):
6.             dp[0][i] = 1      #Fill top row with 1's
7.
8.         for i in range(n):
9.             dp[i][0] = 1      #Fill left column with 1's
10.
11.        for i in range(1, n):
12.            for j in range(1, m):
13.                dp[i][j] = dp[i-1][j] + dp[i][j-1]
14.                #Recursion
15.        return(dp[-1][-1]) #index -1 means last element
16.
17. sol = Solution()
18. print(sol.uniquePaths(7,3))

```

**Output:** 28

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:**

There are two nested loops. Thus, the time complexity is  $O(nm)$ .

### **Space complexity:**

As we are using an array of length  $n \times m$ , the space complexity is  $O(nm)$ .

## **9.11 Question 73 – How many unique paths exist in a square grid having obstacles?**

In the last problem, let us add some obstacles in the grid. The chosen paths should avoid these obstacles.

### **Problem statement**

Here, some cells in the grid contain obstacles. So the robot cannot step on these. The locations of the obstacles are specified by a ‘1’ in the corresponding location in the matrix `obstacleGrid`. Other cells of `obstacleGrid` are 0. `obstacleGrid` is of the same size as the robot walk matrix. Hence,  $m$  and  $n$  can be deduced from it, and they are not provided in the input. Find the number of ways in which the robot can reach from top left to bottom right corner. Let us illustrate this with the help of an example.

Example:

### **Input:**

```
[  
  [0,0,0],  
  [0,1,0],  
  [0,0,0]  
]
```

### **Output:** 2

**Explanation:** There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

### **Solution format**

Your solution should be in the following format:

```
1. class Solution:  
2.     def uniquePathsWithObstacles(self, obstacleGrid:  
List[List[int]]) -> int:
```

## Strategy

We will use the dynamic programming as before. We will create an  $n \times m$  dp array. The content of the element  $dp[i, j]$  represents the number of ways of reaching the cell  $(i, j)$  from the cell  $(0,0)$ .

Consider a cell  $(i, j)$ . As shown in [Figure 9.7](#) in the last question, the incoming paths into this cell are from the top  $(i-1, j)$ , and from the left  $(i, j-1)$ . If any one of them is blocked, then it should not contribute to the path count for the cell  $[i, j]$ . If both of them are blocked, the cell  $[i, j]$  should also be blocked.

This operation can be performed with a lot of ‘if’ statements. But a simple way to implement this logic is to set the blocked cells to 0. The initial value of  $dp[i, j]$  is zero for all  $i$  and  $j$ . However, if you see carefully, we are not using the initial values anywhere in the recurrence relation. So it could be anything. We are using the updated values in the calculation for bigger values of  $i$  and  $j$ . So we will use value 0 to indicate a blocked cell.

If there are  $n_1$  ways of reaching the top and  $n_2$  ways of reaching the left cell, then the number of ways for reaching the cell  $(i, j)$  are  $n_1+n_2$ .

$$dp[i, j] = dp[i-1, j] + dp[i, j-1]$$

Now let us work on the initialization. The top left corner itself could be an obstacle, in which case the robot cannot move at all and we must return 0. If the top left is okay, set  $dp[0][0]$  to 1 and fill the first column. We will iterate over  $dp[i][0]$  with  $i$  going from 1 to  $n-1$ . Unless there is an obstacle, that is,  $obstacleGrid[i][0] = 1$ , we will replicate the upper cell value to the current cell. In case of an obstacle, we will set it to 0. All the next cells down the column will become 0. Similarly, we will fill the top row by iterating over  $dp[0][j]$  with  $j$  going from 1 to  $m-1$ . In case of an obstacle, we will set it to 0. All the next cells to the right will become 0.

Now, we will run two nested loops to implement the recurrence relation stated above and fill the matrix. If there is an obstacle at  $(i, j)$ , then the  $dp[i][j]$  will be made 0. At the end, the answer is contained at the bottom right corner.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def uniquePathsWithObstacles(self, obstacleGrid:
List[List[int]]) -> int:
4.         #print(obstacleGrid)
5.         m = len(obstacleGrid)    # Find number of columns
6.         n = len(obstacleGrid[0])# Find number of rows
7.         dp = [[0]*n for _ in range(m)]# create a 2D-matrix
and initialize with 0
8.         if obstacleGrid[0][0] == 1: return 0
9.         dp[0][0] = 1 # Number of ways of reaching top left
cell = 1.
10.        #Initialize first column
11.        for i in range(1,m):
12.            if obstacleGrid[i][0] == 1:
13.                dp[i][0] = 0
14.            else
15.                dp[i][0] = dp[i-1][0]
16.        #Initialize first row
17.        for j in range(1,n):
18.            if obstacleGrid[0][j] == 1:
19.                dp[0][j] = 0
20.            else
21.                dp[0][j] = dp[0][j-1]
22.        #Scan rest of the matrix
23.        for i in range(1,m):
```

```

24.         for j in range(1,n):
25.             if obstacleGrid[i][j] == 1:
26.                 dp[i][j] = 0
27.             else:
28.                 dp[i][j] = dp[i-1][j] + dp[i][j-1]
29.     return dp[-1][-1]
30. #Driver code
31. sol = Solution()
32. print (sol.uniquePathsWithObstacles([[0,0,0],[0,1,0],
[0,0,0]]))

```

**Output:** 2

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

There are two nested loops. Thus, the time complexity is  $O(nm)$ .

### **Space complexity:**

As we are using an array of length  $n \times m$ , the space complexity is  $O(nm)$ . It is possible to reduce the space complexity to  $O(1)$  by using the `obstacleGrid` matrix itself to store the `dp` information. When we find an `obstacleGrid` to be 1, we will make it 0. Otherwise, replace it with the sum of top and left cells.

## 9.12 Question 74 – What is the longest palindromic substring of a given string?

This question is a standard application of the dynamic programming.

### Problem statement

You are given a string  $s$ . Find the longest substring in  $s$  that is palindromic. Let us illustrate this with the help of two examples.

### **Example 1:**

**Input:** “babad”

**Output:** “bab”

**Explanation:** “bab” is the longest palindromic substring. However, it is not unique. “aba” is also a valid answer.

**Example 2:**

**Input:** “cbbd”

**Output:** “bb”

**Explanation:** “bb” is the longest palindromic substring.

## Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def longestPalindrome(self, s: str) -> str:
```

## Strategy

We will use the dynamic programming philosophy. We will form a substring ss by slicing the string s between the indices i and j. Now we will split the problem into smaller sub-problems of finding the substrings within ss. We will establish a recurrence relation for a length  $n$  palindrome test based on the length  $n-2$  test.

Consider a long string  $s = \dots\dots\dots x \dots\dots\dots x \dots\dots$

Now we will think of a substring of s, named “ss” that is spanned by the indices i and j. The corresponding characters are marked “x” in the strings.

Substring ss = “x\dots\dots\dots x”

Let us define the “inner” substring of ss as the string that is obtained by removing the two extreme x characters. Thus, it is spanned by the indices  $i+1$  and  $j-1$ . What is the condition for ss to be palindromic? We can say that ss is palindromic if the end characters ( $ss[i]$  and  $ss[j]$ ) are identical and the inner substring (that is,  $ss[i+1] \dots ss[j-1]$ ) is palindromic. Let us state the recursion formula as:

ss is palindromic if  $ss[i] = ss[j]$  and inner is palindromic. ... (1)

Note that the empty strings as well as the strings with single alphabet are considered palindromic. Therefore, the exit condition for recursion is when the length of  $s[i:j]$  becomes 0 or 1. We can concisely write the exit condition as:

$$j-i \leq 2$$

We can use either the top-down approach or the bottoms-up approach. In both the cases, we will use the memoisation to avoid the duplication of work. In the top-down approach, we will start with the given string  $s$  and recursively find the palindrome test for smaller substrings. However, it may run into the stack overflow problem. So, a bottoms-up approach with the memoisation is used here.

For exhaustively searching all the substrings, we will use a sliding window approach. The observation window  $s[i:j]$  is defined from  $i$  to  $j$ . We will run an outer loop with the index  $j$  and an inner loop with the index  $i$ . In each iteration, we will examine whether the substring  $s[i:j]$  is palindromic. If yes, we will compare its length with the current maximum length, and if the current maximum length (represented by the indices  $left$  and  $right$ ) is exceeded, it is replaced. When the loops are finished, the longest palindrome substring will be found between the indices  $left$  and  $right$ .

Now, let us design the memoisation strategy. We need to remember the palindromic status of the substrings of all the combinations of  $i$  and  $j$ . For this purpose, we will create an  $n \times n$  memoisation matrix named `palindrome`. If the substring  $s[i:j]$  is palindromic, then `palindrome[i][j]` is 1. Actually, only the upper-half triangle of this matrix is relevant because  $j$  must be greater than  $i$ . As results for the smaller substrings are calculated, they are stored in this matrix and they are reused for the bigger substrings.

The recurrence relation (1) can now be stated in terms of the palindrome matrix as:

$s[i:j]$  is palindromic if `palindrome[i+1][j-1]` is True or when  $j-i \leq 2$  ... (2)

Let us walk through the steps with  $s = "babad"$ . The palindrome matrix will be initialized to zeroes as shown in [Table 9.20](#):

i/j	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

**Table 9.20:** Initial state of the palindrome

### Consider j=1

The variable  $i$  loops through 0 to  $j-1$ , that is, only 0.  $ss = s[0:1] = "ba"$ .  $s[i]=b$  and  $s[j]=a$ . As these are not equal, the  $\text{palindrome}[0][1]$  remains 0.

### Consider j=2

The variable  $i$  loops through 0 to  $j-1$ , that is, 0 and 1.

When  $i = 0$ ,  $ss = "bab"$ ,  $s[i]=b$  and  $s[j]=b$ . The inner string ("a") is palindromic because  $j-i = 2$ . Moreover,  $s[i]=s[j]=b$ , the substring  $ss$  is palindromic. Make  $\text{palindrome}[0][2] = \text{True}$ . Since we have found a valid palindrome, update the indices left and right.

`left = 0, right = 2.`

When  $i = 1$ ,  $ss = "ab"$ ,  $s[i]=a$ , and  $s[j]=b$ . As these are not equal, the  $\text{palindrome}[1][2]$  remains 0.

### Consider j=3

The variable  $i$  loops through 0 to  $j-1$ , i.e., 0 to 2.

When  $i = 0$ ,  $ss = "baba"$ ,  $s[i]=b$ , and  $s[j]=a$ . As these are not equal, the  $\text{palindrome}[0][3]$  remains 0.

When  $i = 1$ ,  $ss = "aba"$ ,  $s[i]=a$ , and  $s[j]=a$ . As  $s[i]=s[j]=b$ , and the inner substring b is a single character,  $ss$  is palindromic. Make the  $\text{palindrome}[1][3] = \text{True}$ . The length of the new palindrome (3) is not bigger than the current maximum. So, leave the left and the right untouched.

When  $i = 2$ ,  $ss = "ba"$ ,  $s[i]=b$ , and  $s[j]=a$ . As these are not equal, the  $\text{palindrome}[2][3]$  remains 0.

### Consider j=4

The variable  $i$  loops through 0 to  $j-1$ , i.e., 0 to 3. The corresponding substrings are, “babad”, “abad”, “bad”, “ad”.

We do not find any new palindromes here.

The final state of the palindrome matrix is shown in [Table 9.21](#):

i/j	0	1	2	3	4
0	0	0	T	0	0
1	0	0	0	T	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

*Table 9.21: Final state of palindrome*

It shows two true values. The first one is at (0,2) corresponding to bab. The second one is at (1,3) corresponding to aba.

### Python code

The following code implements this strategy:

```
1. class Solution:  
2.     def longestPalindrome(self, s: str) -> str:  
3.         n = len(s)  
4.         if(n<2):  
5.             return s  
6.         left = 0      #Start of longest substring  
7.         right = 0  
8.  
9.         palindrome = [[0]*n for _ in range(n)]  
10.  
11.        for j in range(1,n):
```

```

12.         for i in range(0,j):
13.             innerIsPalindrome = palindrome[i+1][j-1] or
j-i<=2
14.             si=s[i]
15.             sj=s[j]
16.             if(s[i] == s[j] and innerIsPalindrome):
17.
18.                 palindrome[i][j] = True
19.                 if(j-i>right-left):
20.                     left = i
21.                     right = j
22.                 print(palindrome)
23.             return s[left:right+1]
24. sol = Solution()
25. print (sol.longestPalindrome('babad'))

```

**Output:** bab

## Complexity Analysis

The complexity analysis is as follows:

**Time complexity:** There are two nested loops. Thus, the time complexity is  $O(n^2)$ .

**Space complexity:** As we are using an array of length  $n \times n$ , the space complexity is  $O(n^2)$ .

## 9.13 Question 75 – How much rain water can be trapped in the ridges?

The dynamic programming technique can be used to solve this abstract problem.

### Problem statement

Consider an imaginary terrain consisting of the long ridges having flat-shaped tops and stair-shaped slopes. The width and height of the stairs is

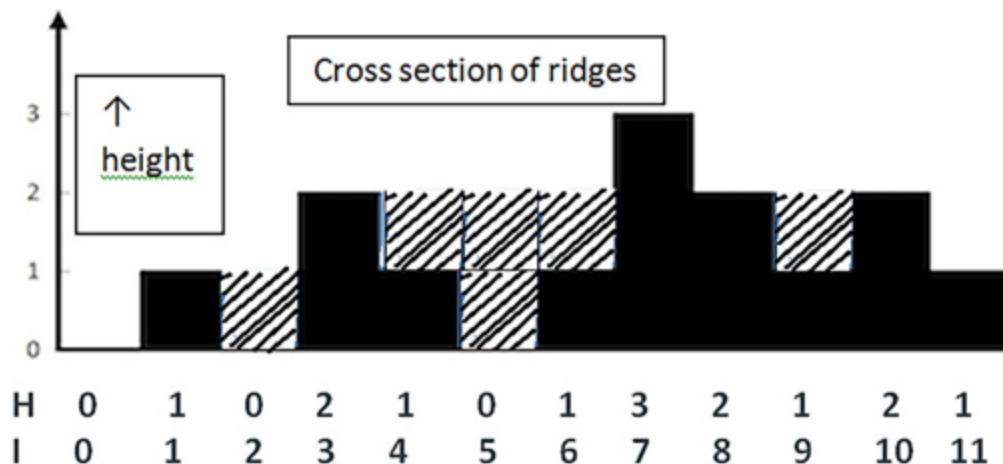
fixed to 1 unit. The height of the ridges is tabulated in the array heights. Now, suppose there is a good rainfall on this terrain. Find how much rain water will be trapped per unit of length on the ridges. Let us illustrate this with the help of an example.

### Example 1:

**Input:** heights = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The cross-section of the terrain is shown [Figure 9.8](#). The water trapped after rainfall is shown by the hatched lines. The water is trapped in ridges 2, 4, 5, 6, and 9.



*Figure 9.8: Cross section of ridges*

Total amount of water =  $1 + 1 + 2 + 1 + 1 = 6$ .

### Solution format

Your solution should be in the following format:

```
1. class Solution:  
2.     def trap(self, height: List[int]) -> int:
```

### Strategy

Let us understand the problem precisely by thinking in the brute force manner and then we will apply the dynamic approach. Let us find the water trapped in each bin of the heights array shown in [Figure 9.8](#).

The heights[0] cannot store any water because there is nothing to hold it on the left side.

The heights[1] cannot store any water because there are no higher barriers on the either side. The heights[2] can store 1 unit of water because it is flanked by a ridge of 1 unit height on the left side and a ridge of 2 units of height on the right side. The minimum of the two is 1 unit.

The heights[3] cannot store any water because there are no higher barriers on either side. The heights[4] can store 1 unit of water because it is flanked by a ridge of 2 units height on the left side and a ridge of 3 units of height on the far right side and its bottom height is 1 unit.

The heights[5] can store 2 units of water because it is flanked by a ridge of 2 units height on the left side and a ridge of 3 units of height on the far right side and its bottom height is 0.

Now, we can generalize the formula for water capacity of the bin  $i$ :

$$\text{water}[i] = \min(\text{left\_max}[i], \text{right\_max}[i]) - \text{heights}[i] \dots (1)$$

Where  $\text{left\_max}[i]$  is the maximum ridge height on the left side of  $i$  and  $\text{right\_max}[i]$  is the maximum ridge height on the right side of  $i$ . The  $\text{heights}[i]$  is the height of the bottom.

In the brute force method, we will scan the bins from 0 to  $n$ , and add water in each bin to total. We need to calculate  $\text{left\_max}[i]$  and  $\text{right\_max}[i]$  in every iteration. Here comes the dynamic programming and memoisation. We will build an array  $\text{left}$  whose  $i^{\text{th}}$  element will contain the maximum height to the left of  $i$ . The recurrence relation to find  $\text{left}[i]$  in terms of earlier values is:

$$\text{left}[i] = \max(\text{left}[i-1], \text{height}[i])$$

The filled left array will look as follows:

$$\text{left} = [0, 1, 1, 2, 2, 2, 3, 3, 3, 3]$$

Similarly, we will build an array  $\text{right}$  whose  $i^{\text{th}}$  element will contain the maximum height to the right of  $i$ . For building this array, we will have to start from the rightmost element and work backwards. The recurrence relation is:

$$\text{right}[i] = \max(\text{right}[i+1], \text{height}[i])$$

The filled right array will look as follows:

```
right =[3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 1]
```

Now, it is a simple matter to sum up the individual water capacity over all the elements of heights.

## Python code

The following code implements this strategy:

```
1. from typing import List
2. class Solution:
3.     def trap(self, height: List[int]) -> int:
4.         n = len(height)
5.         if(n == 0):
6.             return 0
7.         #Create n element memoisation arrays left and right
8.         left = [0]*n
9.         right = [0] * n
10.        ans = 0 #Initialize answer to 0
11.        #Prepare left array
12.        left[0] = height[0]
13.        for i in range(1, n):
14.            left[i] = max(left[i-1], height[i])
15.        #Prepare right array
16.        right[n-1] = height[n-1]
17.        for i in range(n-2, -1, -1):
18.            right[i] = max(right[i+1], height[i])
19.        #Sum up water
20.        for i in range(0, n):
21.            ans += min(left[i], right[i])-height[i]
22.        return ans
23. sol = Solution()
24. print(sol.trap([0,1,0,2,1,0,1,3,2,1,2,1]))
```

**Output:** 6

## Complexity Analysis

The complexity analysis is as follows:

### **Time complexity:**

We run 3 loops of length  $n$ . Thus, the time complexity is  $O(3n)$ . We can call it  $O(n)$ .

### **Space complexity:**

As we are using an array of length  $n$ , the space complexity is  $O(n)$ .

## Conclusion

In this chapter, you were introduced to an important algorithmic technique, namely dynamic programming. We studied 12 questions covering a variety of situations where dynamic programming can be applied. We started with the basic principles of dynamic programming, i.e., breaking a problem into smaller sub-problems and storing the results of sub-problems for future reuse. In question 64, we solved the knapsack problem where an item was not allowed to be split. Here, we created a series of problems from capacity of 1 KG to 5 Kg and solved each one incrementally. In question 65, to find the maximum sales, we created a series of problems, starting from 1 house to  $n$  houses. We followed the same strategy from questions 66 to 71. In questions 72 and 73, we extended the philosophy to two dimensions. Question 74 was somewhat different. The trick was to find a proper recurrence relation. We found a palindrome test of the  $n$ -long string in terms of  $(n-2)$  long substring. In question 75 on the water storage required splitting of the problem into smaller problems and cleverly reusing the intermediate results.

Most questions on dynamic programming can be mapped to the concepts presented in this chapter.

With this we come to the end of the book. By now you must have developed the capability of recognizing a problem type and applying the appropriate algorithmic technique. You must have learned the special features of Python that make the solution concise, elegant, and efficient. You should now be standing firmly on your feet to face any unseen challenge in future.

## Points to Remember

- Dynamic programming is an optimization technique.
- The two basic requirements of dynamic programming are optimal substructure and overlapping sub-problems.
- The process of storing sub-problem results for future use is called memoisation.
- The solutions can be built in top-down or bottoms-up manner.
- `@lru_cache` prefix can be used for automatic memoisation

## MCQs

1. Which of the following properties are needed in a dynamic programming problem?
  - A. Optimal substructure
  - B. Overlapping sub problems
  - C. Greedy approach
  - D. Both optimal substructure and overlapping sub problems
2. If an optimal solution can be created for a problem by constructing optimal solutions for its sub-problems, the problem possesses \_\_\_\_\_ property.
  - A. Overlapping sub problems
  - B. Optimal substructure
  - C. Memoization
  - D. Greedy
3. If a problem can be broken into sub problems which are reused several times, the problem possesses \_\_\_\_\_ property.
  - A. Overlapping sub-problems
  - B. Optimal substructure
  - C. Memoization
  - D. Greedy

4. If a problem can be solved by combining the optimal solutions to non-overlapping problems, the strategy is called \_\_\_\_\_.
- Dynamic programming
  - Greedy
  - Divide-and-conquer
  - Recursion
5. In dynamic programming, the technique of storing the previously calculated values is called \_\_\_\_\_.
- Saving value property
  - Storing value property
  - Memoization
  - Mapping
6. When a top-down approach of dynamic programming is applied to a problem, it usually \_\_\_\_\_.
- Decreases both, the time complexity and the space complexity
  - Decreases the time complexity and increases the space complexity
  - Increases the time complexity and decreases the space complexity
  - Increases both, the time complexity and the space complexity

## Answers to MCQs

**Q1:** D

**Q2:** B

**Q3:** A

**Q4:** C

**Q5:** C

**Q6:** B

## Questions

1. Quote an example where the dynamic programming algorithm does not work.
2. What properties are essential in a problem for the dynamic programming to be useful?
3. How is automatic memoisation achieved with the `@lru_cache` function?
4. Catalan numbers are generated by the following recursive formula.

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} \quad \text{Where } C_0 = 1$$

For example:

$$C_1 = C_0 * C_0 = 1$$

$$C_2 = C_0 * C_1 + C_1 * C_0 = 1 * 1 + 1 * 1 = 2$$

$$C_3 = C_0 * C_2 + C_1 * C_1 + C_2 * C_0 = 1 * 2 + 1 * 1 + 2 * 1 = 5$$

$$C_4 = C_0 * C_3 + C_1 * C_2 + C_2 * C_1 + C_3 * C_0 = 1 * 5 + 1 * 2 + 2 * 1 + 5 * 1 = 14$$

Write a top-down dynamic programming solution with the memoisation for finding the  $n^{\text{th}}$  Catalan number.

## Key terms

Dynamic programming, memoisation, optimal substructure, overlapping sub-problems

# Index

## Symbols

0/1 knapsack  
value, maximizing [334-337](#), [339](#)

## A

abstract optimization  
about [35](#)  
approaches [37](#)  
problem, solving [35](#), [36](#)  
approaches, abstract optimization  
brute force approach [37](#)  
pinching window approach [37](#), [38](#)  
approaches, array  
complexity analysis [29](#)  
Pseudo code [28](#)  
Python code [28](#)  
approaches, array duplicates  
brute force method [123](#), [124](#)  
hash table, using [126](#), [127](#)  
set method, using [125](#), [126](#)  
sorting [124](#), [125](#)  
approaches, array element  
Counter method, using [132](#), [133](#)  
count method, using [131](#)  
from basics [130](#)  
get method, using [131](#), [132](#)  
approaches, array validation  
complexity analysis [35](#)  
Python code [34](#)  
approaches, integer  
arithmetic progression [25](#), [26](#)  
linear search [24](#)  
approaches, k closest points  
selection sort [320-323](#)  
sorting [319](#), [320](#)

approaches, math-oriented problem  
brute force method [137](#), [138](#)  
hash table [138](#), [139](#)

approaches, of binary search in sorted array  
complexity analysis [41](#)  
Python code [40](#)

approaches, people array  
complexity analysis [32](#)  
Python code [31](#)

approaches, prime numbers  
brute force method [140](#), [141](#)  
Eratosthenes sieve method [141](#)-[143](#)

approaches, search target  
2D binary search [74](#), [75](#)  
2D linear search [72](#), [74](#)  
linear search [72](#)

approaches, single number  
algebraic method [155](#)  
hash table, using [154](#)  
list, using [153](#)

approaches, sorted array  
binary search [20-22](#), [66-70](#)  
linear search [18](#), [19](#)  
linear search (brute force) [65](#), [66](#)

approaches, square root  
binary search [63](#), [64](#)  
linear search [62](#), [63](#)

approaches, stack minimum implementation  
space complexity O(1) [113](#), [114](#)  
time complexity O(n) [112](#), [115](#)

approaches, stairs climbing  
Binet's formula [359](#), [360](#)  
compact array [355](#), [356](#)  
dynamic programming [354](#), [355](#)  
matrix exponentiation [357](#), [358](#)

approaches, stock market  
brute force method [344](#), [345](#)  
dynamic programming [345](#)-[347](#)

approaches, string brackets  
complexity analysis [136](#)  
Pseudo code [135](#)

Python code [135](#)  
approaches, subarray  
    brute force method [55-58](#)  
    efficient method [58-60](#)  
approaches, subsets  
    binary representation [44-47](#)  
    output array [42-44](#)  
approaches, substring  
    brute force method [144, 145](#)  
    sliding window [146-149](#)  
approaches, symmetric binary tree  
    iterative [202, 203](#)  
    recursive [200, 201](#)  
array  
    approaches [27](#)  
    function [27](#)  
    program, writing [26, 27](#)  
    two-pointer sliding window technique [26](#)  
array duplicates  
    approaches [123](#)  
    finding [122, 123](#)  
    finding, in vicinity [127-129](#)  
array element  
    approaches [130](#)  
    finding [129, 130](#)  
array validation  
    about [32](#)  
    approaches [34](#)  
    function, writing [32, 33](#)  
    solution [34](#)

## B

backtracking, objectives  
    decision [278](#)  
    enumeration [279](#)  
    optimization [279](#)  
backtracking principle [278](#)  
backtracking problems  
    about [279](#)  
    Hamiltonian circuit [279](#)

N-Queens [279](#)  
balanced binary tree  
    validating [218-220](#), [222](#)  
BFS graph class  
    basics [248](#), [249](#)  
    recursive approach [250](#)  
binary search [15](#)  
binary search, in sorted array  
    about [38](#)  
    approaches [39](#), [40](#)  
    faulty version [39](#)  
    solution [39](#)  
binary search tree  
    Kth smallest element [209-211](#), [213](#)  
    validating [222-225](#), [227](#)  
binary search tree (BST)  
    about [293](#)  
    elements, within range [293-295](#), [297](#)  
binary tree  
    approaches [204](#), [205](#)  
    lowest common ancestor, finding [174-176](#), [178](#)  
    maximum height [203](#), [204](#), [206](#)  
    maximum path  
        sum, finding [213-215](#), [218](#)  
    representation, in memory [165](#), [166](#)  
binary tree, path sum  
    approaches [207](#), [209](#)  
    finding [206](#)  
bridge  
    building, to connect islands [268-274](#)

## C

coin patterns  
    to dispense change [368-373](#)  
complexity function  
    O(1) [4](#)  
    O(2n) [10](#), [11](#)  
    O(n) [4](#)  
    O(n<sup>2</sup>) [5](#), [6](#)  
    O(n<sup>3</sup>) [6-9](#)

$O(n \log n)$  [9](#), [10](#)  
course schedule  
    validating [235-239](#)  
course sequence  
    validating [260-263](#)

## D

defaultdict  
    using [121](#)  
DFS algorithm  
    running [189](#)  
DFS Traversal of Graphs [188-192](#)  
DFS Traversal of Trees  
    about [188-193](#)  
    approaches [193](#)  
divide-and-conquer strategy  
    applications [305](#)  
    used, for sorting [315-318](#)  
Docstring [27](#)  
door-to-door salesman  
    dynamic programming principle, applying [339-343](#)  
dynamic programming  
    approach [329](#)  
    principles [328](#), [330-334](#)  
dynamic programming, properties  
    optimal substructure property [328](#)  
    overlapping sub-problems [329](#)

## E

Eratosthenes sieve [141](#)  
execution time  
    measuring [47](#)

## F

function  
    writing, to remove island [232](#), [233](#), [235](#)

## G

graph  
manipulation [162-164](#)  
plotting, for sort function [51, 52](#)  
plotting, for subsets of set problem [53, 54](#)  
representation [162, 163](#)

Graph Theory  
basics [160, 161](#)

graph theory based model  
formulating [255-258, 260](#)  
greedy algorithm [313-315](#)  
greedy methodology, problem  
about [304, 305](#)  
Dijkstra's Shortest Path [305](#)  
Huffman Coding [305](#)  
Kruskal's Minimum Spanning Tree (MST) [304](#)  
Prim's Minimum Spanning Tree [305](#)

grid  
word, searching in [279-281, 283](#)

## H

Hamiltonian circuit [279](#)  
hash table  
implementing, in Python [120, 121](#)

## I

integer  
approaches [23](#)  
function [23](#)  
function, writing [23](#)  
validating [23](#)  
integer, converting into roman numeral  
about [75, 76](#)  
approaches [77, 78](#)  
solution [76](#)

## K

k closest points  
approaches [319](#)  
finding [318](#)

k digits  
removing, from number [309-312](#)  
knapsack  
value maximizing [306, 307, 309](#)  
k subsets  
set, partitioning into [297-301](#)  
Kth smallest element  
in binary search tree [209-211, 213](#)

## L

linear data structures, in Python  
about [11, 12](#)  
list class [12, 13](#)  
NumPy arrays [14](#)  
strings [14](#)  
linear search [15](#)  
linked list  
adding [99-101](#)  
approaches [94, 95](#)  
basics [86](#)  
creating [87, 88](#)  
creating, from array [89, 90](#)  
cycle, detecting [93](#)  
cycle detect, solution [94](#)  
displaying [87, 88](#)  
even node, creating [104-107](#)  
node [86](#)  
node, adding at end [89](#)  
node, adding at start [88](#)  
odd node, creating [104-107](#)  
reversing [96-98](#)  
list class [12, 13](#)

## M

map coloring problem  
about [181, 182](#)  
approaches [182-185](#)  
math-oriented problem  
approaches [137](#)

finding [136](#), [137](#)  
maze  
traversing [283-286](#)

## N

nCk combinations  
finding [287-289](#)  
node [86](#)  
node, data structure  
information [86](#)  
link [86](#)  
N-Queens [279](#)  
nth node  
removing [101-104](#)  
number of coins  
to dispense change [365-368](#)  
NumPy arrays [14](#)  
nums [26](#)

## P

palindromic substring [380-384](#)  
partitioning [316](#)  
Pascal's triangle, construction  
about [78](#), [79](#)  
approaches [79-81](#)  
program, writing [79](#)  
solution [79](#)  
people array  
about [29](#)  
approaches [30](#)  
problem, solving [29](#), [30](#)  
solution [30](#)  
perf\_counter  
using [48](#)  
power set [41](#)  
prime numbers  
approaches [140](#)  
counting [140](#)  
Python

hash table, implementing in [120](#), [121](#)

linear data structures [11](#), [12](#)

## R

recursion

about [169](#)

output [170](#)

redundant edge

approaches [171-174](#)

detecting [170](#), [171](#)

reverse polish postfix expression

evaluating [108-111](#)

ridges

need for [385-388](#)

roman numeral

converting, into decimal numeral [149-152](#)

## S

sales manager

rewarding [240-243](#)

searching

about [15](#)

purpose [15](#), [16](#)

search target

approaches [72](#)

in 2D matrix [71](#)

program, writing [71](#)

solution [71](#)

set

partitioning, into k subsets [297-301](#)

set()

using [122](#)

single number

approaches [153](#)

identifying, in array [152](#)

sorted array

approaches [18](#), [65](#)

automatic evaluation platform, testing [17](#)

target number [64](#)

target position [16](#)  
target value, finding [64](#)  
target value, searching [16](#)  
target value, solution [65](#)  
sorted lists  
    merging [90](#)  
    merging, solution [91](#)  
    node class, object creating [91, 93](#)  
sort function  
    graph, plotting for [51, 52](#)  
sorting [15](#)  
space complexity  
    about [2-4](#)  
    concept [11](#)  
square grid  
    unique paths [373-376](#)  
    unique paths obstacles [377-380](#)  
square root  
    about [61](#)  
    approaches [61](#)  
    finding [61](#)  
    solution [61](#)  
stack minimum implementation  
    achieving, with list [111, 112](#)  
    approaches [112](#)  
stairs climbing  
    about [351](#)  
    approaches [352, 353, 361-364](#)  
    cost, minimizing [360, 361](#)  
standard DFS technique  
    using [228-230, 232](#)  
state-space tree [278](#)  
stock  
    buy and sell [347-351](#)  
stock market  
    approaches [344](#)  
    dynamic programming application, studying in [343, 344](#)  
string  
    partitioning, into palindrome segments [289-291, 293](#)  
string brackets  
    approaches [135](#)

validation, finding [133](#), [134](#)  
strings [14](#)  
subarray  
    approaches [55](#)  
    maximum sum, finding [54](#), [55](#)  
    solution [55](#)  
subsets  
    about [41](#)  
    approaches [42](#)  
    finding [41](#)  
    solution [42](#)  
subsets of set problem  
    graph, plotting for [53](#), [54](#)  
substring  
    finding, without characters repeated [143](#), [144](#)  
symmetric binary tree  
    approaches [199](#)  
    validating [198](#), [199](#)

## T

tickets itinerary  
    reconstructing [194-198](#)  
time complexity  
    about [2-4](#)  
    measuring [47](#)  
time complexity graphs  
    plotting [50](#), [51](#)  
timeit function  
    arguments [49](#)  
    using [49](#)  
town judge  
    about [178](#)  
    approaches [179-181](#)  
tree  
    creating, as input data [166-168](#)  
    displaying, as input data [166](#), [168](#)  
    manipulation [165](#)  
    representation [165](#)  
tree level-wise list  
    printing [251-254](#)

tree nodes

testing [263](#), [264](#)

validating [265](#), [267](#)