

Data Structures & Algorithms



For Job Interviews

A. J. García

Data Structures and Algorithms for Job Interviews

Prep for the interview and get the job you want

Alejandro Garcia

This book is for sale at <http://leanpub.com/data-structures-algorithms-for-job-interviews>

This version was published on 2020-08-15



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Alejandro Garcia

Contents

Other Books by Alejandro	1
Recommended Resources	4
Introduction	6
Who is this book for ?	6
What this book covers ?	6
Chapter 1: Bit Manipulation	8
Check whether a given number n is a power of 2 or 0	8
Count number of bits needed to be flipped to convert A to B	8
Find the two non-repeating elements in an array of repeating elements	9
Find the next greater and next smaller number with same number of set bits	10
Chapter 2: Dynamic Programming	12
0-1 Knapsack Problem	12
Cutting Rod problem	13
Minimum number of edits (operations) required to convert 'str1' into 'str2'	14
Given a 2-D matrix of 0s and 1s, find the Largest Square which contains all 1s in itself	15
Given two sequences, print the longest subsequence present in both of them.	16
Length of the longest subsequence in an array such that all elements of the subsequence are sorted in increasing order	18
Find minimum cost path in a matrix from (0,0) to given point (m,n)	18
Partition a set into two subsets such that the difference of subset sums is minimum	19
Minimum number of umbrellas of m different sizes required to accomodate N people	20
Determine if there is a subset of the given set with sum equal to given sum	22
Given a distance 'dist, count total number of ways to cover the distance with 1, 2 and 3 steps	22
Chapter 3: Graph	24
Find all possible words in a board of characters	24
Breadth First Search Traversal	25
Depth First Search Traversal	27
Detect Cycle in directed graph	29
Detect cycle in undirected graph	30
Dijkstra's Shortest Path Algorithm	32

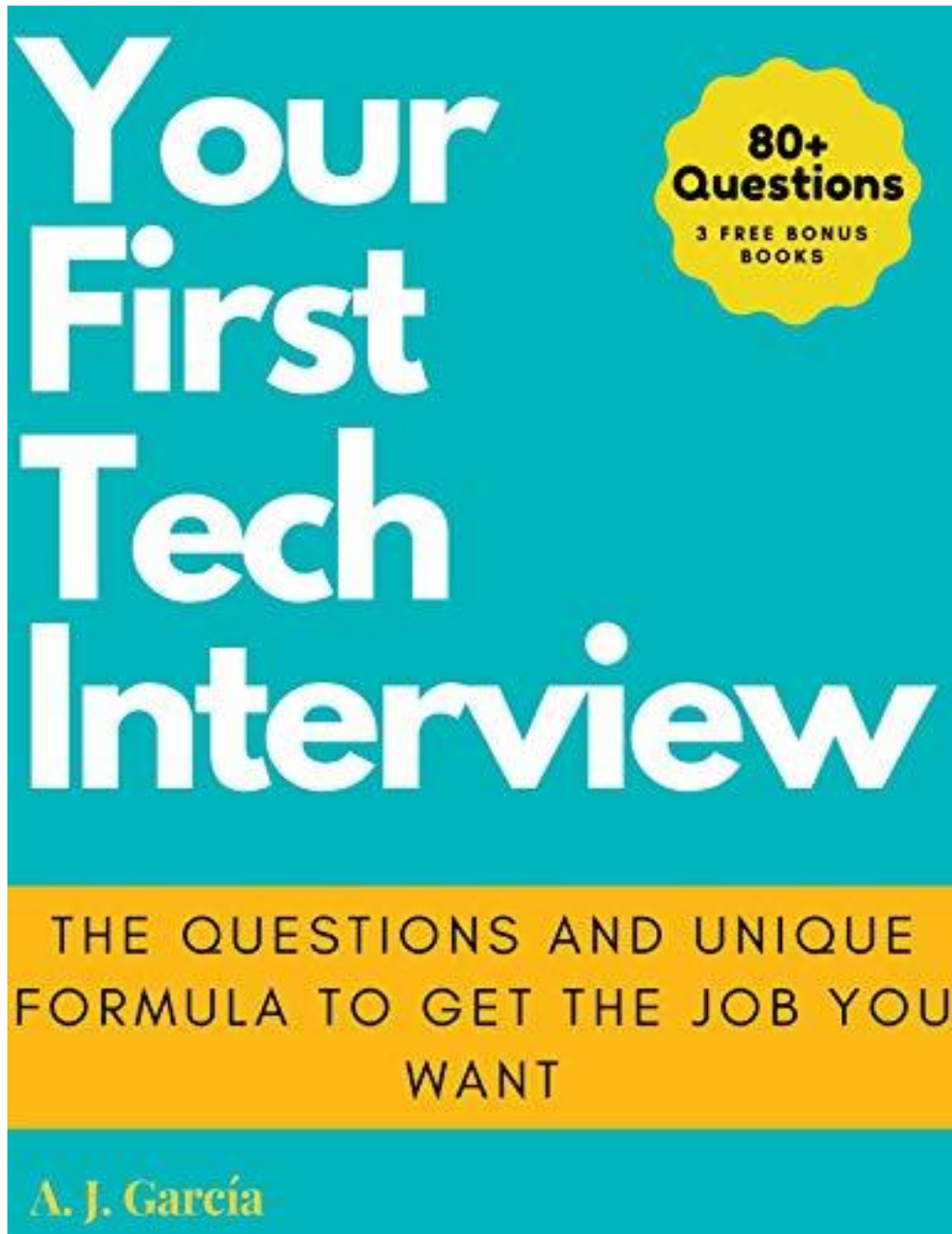
CONTENTS

Find shortest distances between each pair of vertices in a given edge weighted directed Graph	34
Graph implementation	36
Kruskal's Algorithm for Minimum Spanning Tree	38
Topological Sort	41
Chapter 4: Heaps	43
Heap Sort algorithm	43
Max Heap implementation	44
Min Heap implementation	47
Find the median for an incoming stream of numbers after each insertion in the list of numbers	50
Chapter 5: Linked List	57
Clone a linked list with next and random pointer	57
Given a linked list of line segments, remove middle points if any	60
Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes.	62
Merge a linked list into another linked list at alternate positions	65
Perform Merge Sort	67
Point to next higher value node in a linked list with an arbitrary pointer	71
Find if linked list contains any cycle	75
To select a random node in a singly linked list	77
Find and remove cycle if any	78
Reverse alternate sub list of K nodes each	82
Reverse a linked list	84
Bring even valued nodes to the front	86
Implementation of Singly Linked List	89
Chapter 6: Mathematics	94
Find the number of trailing zeros in factorial of a number	94
Find the greatest common divisor of 2 numbers	94
Print all prime factors of a given number	94
Sieve of Eratosthenes (find prime numbers up to n efficiently)	95
Chapter 7: Matrix	97
Given the Coordinates of King and Queen on a chessboard, check if queen threatens the king	97
Search in a row wise and column wise sorted matrix	97
Given a 2D array, print it in spiral form	98
Chapter 8: Strings or Arrays	100
Find the longest substring with k unique characters in a given string	100
Find a pattern in a string using KMP search algorithm	101

CONTENTS

Find the Kth smallest element in the array	103
Find a pair in an array with sum x	104
Print all valid (properly opened and closed) combinations of n pairs of parentheses	105
Reverse the order of the words in the array	106
Find index of given number in a sorted array shifted by an unknown offset	107
Print all permutations of a given string	108
Linear Search in an array	109
Binary Search in an array	110
Interpolation Search in an array	110
Bubble sort Algorithm	111
Counting sort Algorithm (non-comparision based sorting)	112
Insertion sort Algorithm	113
Sort an array where each element is at most k places away from its sorted position	113
Merge Sort Algorithm	114
Quick Sort Algorithm using last element as pivot	116
Selection sort Algorithm	117
Chapter 9: Tree	118
Binary Search Tree implementation	118
Check if a given array can represent Preorder Traversal of Binary Search Tree	121
Find the in-order ancestor of a given node in BST	122
Find the Lowest Common Ancestor	123
Given a sorted array, create a BST with minimal height	125
Print Nodes in Bottom View of Binary Tree	126
Check if a binary tree is height balanced	127
Check whether a binary tree is a full binary tree or not	128
Given two binary trees, check if the first tree is subtree of the second one	130
Find the Lowest Common Ancestor in a Binary Tree	131
Create a list of all nodes at each depth	134
Find the maximum path sum i.e. max sum of a path in a binary tree	135
Find minimum depth of a binary tree	136
Remove nodes on root to leaf paths of length < K	137
Given a Perfect Binary Tree, reverse the alternate level nodes of the tree	138
Print Nodes in Top View of Binary Tree	140
Implementation of Trie data structure	141
Keep developing your programming skills	146
About the Author	147

Other Books by Alejandro



Your First Tech Interview: The Questions and Unique Formula to get the Job you want

Your First Tech Interview: The Questions and Unique Formula to get the Job you want¹



DevOps: Ultimate Beginners Guide to become a Successful Devops Engineer

DevOps: Ultimate Beginners Guide to become a Successful Devops Engineer²

¹https://www.amazon.com/Your-First-Tech-Interview-Questions-ebook/dp/B08DN1BX2V/ref=sr_1_3?crd=2WMGG8O7DREFP&dchild=1&keywords=tech+interview+questions&qid=1596198458&s=books&sprefix=tech+intervi%2Caps%2C261&sr=1-3

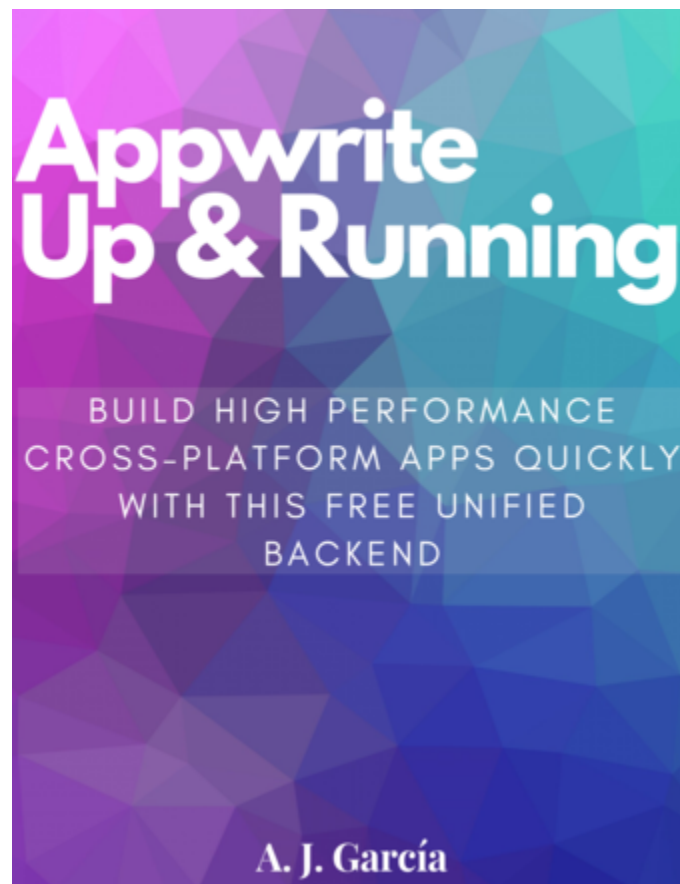
²https://leanpub.com/devops-engineer/?utm_source=ebook&utm_medium=leanpub&utm_campaign=datstruct



Javascript Snippets: Secret Code Snippets used by Powerful Developers to create Innovative Apps

Javascript Snippets: Secret Code Snippets used by Powerful Developers to create Innovative Apps³

³https://leanpub.com/javascript-snippets/?utm_source=ebook&utm_medium=leanpub&utm_campaign=datstruct



Appwrite Up and Running: Build High Performance cross-platform Apps quickly with this Free unified Backend

Appwrite Up and Running: Build High Performance cross-platform Apps quickly with this Free unified Backend⁴

Recommended Resources

If you are looking into video tutorials and trainings to polish your Programming skills take a look at the below options:

- [Udemy Data Structures](#)⁵
- [Data Structures](#)⁶

⁴https://leanpub.com/appwrite-up-and-running/?utm_source=ebook&utm_medium=leanpub&utm_campaign=datstruct

⁵<https://click.linksynergy.com/deeplink?id=NcMZU1aTATA&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourses%2Fsearch%2F%3Fq%3Ddata%2Bstructures>

⁶<https://click.linksynergy.com/deeplink?id=NcMZU1aTATA&mid=40328&murl=https%3A%2F%2Fwww.coursera.org%2Flearn%2Fdata-structures>

Tuesday Messages

On Tuesday, I send some of my Free books to subscribers. You can expect updates, technical articles and courses recommendations. People tell me this content is so valuable to them that sometimes they ask me to send info on particular subjects.

If you would like to **receive the exclusive free content only for subscribers**, then you can join here: [Alejandro's Updates](https://alejandro-garcia-author.ck.page/2ed39ffaf)⁷

⁷<https://alejandro-garcia-author.ck.page/2ed39ffaf>

Introduction

Congrats, if you are reading this I assume you are starting to prepare for your next interview !!

Getting an interview is the first step towards your new life, every Software Engineer has some interesting stories about the interview that landed their current job, some are like a fairy tale (magical and exciting) while others are more like a day in the battlefield behind enemy lines.

The truth is, you have to be prepared and I'm here to help you with that. I've put together a list of questions and answers for you to start practicing your next interview. The more questions you get to study and think of alternative answers the higher chance of success.

In no way this book guarantees you are getting these exact questions, but they are a good way for you to start building the confidence required to go through the interview process.

I'm happy to say that I'm putting together a series of books for Software Engineers, Developers and IT professionals on how to better prepare and deal with the interview questions.

When I wrote *"Your First Tech Interview: The Questions and Unique Formula to get the Job you want"* my intention was to provide a guide and strategy on how to approach the interview process so at the end your chances of getting the job are close to 99.9%.

After completing that book, I thought about writing a series of books for other areas with additional questions. **I hope all these books help you on the path to your new career.**

Who is this book for ?

This book is for those Software Engineers and Developers who are artists using code to bring to life the most powerful systems, it's a quick reference with interview questions and answers for them to practice until they achieve perfection in their interviews.

The interview panel will be trying to make a critical decision, they need to bring onboard a new Software Developer that will help them improve the efficiency of their applications. Your job now is to review as many possible questions as you can so then you ignite amazing conversations revealing innovative ways of optimizing the performance of those apps.

What this book covers ?

The book is a collection of Python code solving some of the common Data Structures and Algorithms you might be expected to solve at an interview process, the book is splitted in 5 different chapters covering the following Data Structures and Algorithms:

Chapter 1 Bit Manipulation.

Chapter 2 Dynamic Programming.

Chapter 3 Graph.

Chapter 4 Heaps.

Chapter 5 Linked List.

Chapter 6 Mathematics.

Chapter 7 Matrix.

Chapter 8 Strings or Arrays.

Chapter 9 Tree.

Chapter 1: Bit Manipulation

Check whether a given number n is a power of 2 or 0

```
1  # Check whether a given number n is a power of 2 or 0
2
3
4  def check_pow_2(num):
5      if num == 0:
6          return 0
7
8      if num & (num - 1) == 0:
9          return 1
10
11     return -1
12
13
14  switch = {
15      0: "Number is 0",
16      1: "Number is a power of 2",
17      -1: "Number is neither a power of 2 nor 0"
18  }
19  case = check_pow_2(16)
20
21  print(switch[case])
```

Count number of bits needed to be flipped to convert A to B

```
1  # Count number of bits needed to be flipped to convert A to B
2
3
4  def count_bits_flip(a, b):
5      # XOR a and b to get 1 on opposite value bit position
6      c = a ^ b
7
8      # initialise the counter for 1
9      count = 0
10
11     # count the number of 1s while there is 1 in a ^ b
12     while c != 0:
13         count += 1
14         c &= (c-1)
15
16     # return the count of 1s
17     return count
18
19
20 # 2 = 0010
21 # 8 = 1000
22 print(count_bits_flip(2, 8))
```

Find the two non-repeating elements in an array of repeating elements

```
1  # Find the two non-repeating elements in an array of repeating elements
2
3
4  def find_non_repeating_numbers(arr):
5      xor = arr[0]
6
7      for i in range(1, len(arr)):
8          xor ^= arr[i]
9
10     right_set_bit = xor & ~(xor-1)
11     first = 0
12     second = 0
13     for i in arr:
14         if i & right_set_bit:
15             first ^= i
```

```
16         else:
17             second ^= i
18
19     return first, second
20
21
22 arr = [2, 3, 7, 9, 11, 2, 3, 11]
23 print(find_non_repeating_numbers(arr))
```

Find the next greater and next smaller number with same number of set bits

```
1  # Find the next greater and next smaller number with same number of set bits
2
3  # Ex. num = 6 bin = 110
4  def next_greater(num):
5      res = num
6      if num != 0:
7          # Find the right most 1 position
8          # Ex. right_one = 2 bin = 10
9          right_one = num & -num
10
11         # get the left pattern to merge
12         # Ex. left_pattern = 8 bin = 1000
13         left_pattern = num + right_one
14
15         # get the right pattern to merge
16         # Ex. right_pattern = 1 bin = 0001
17         right_pattern = (num ^ left_pattern) >> (right_one + 1)
18
19         # OR both the patterns
20         # Ex. res = 9 bin = 1001
21         res = left_pattern | right_pattern
22
23     return res
24
25
26 def next_smaller(num):
27     return ~next_greater(~num)
28
29
```

```
30 print(next_greater(6))  
31 print(next_smaller(6))
```


Chapter 2: Dynamic Programming

0-1 Knapsack Problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays $val[0..n-1]$ and $wt[0..n-1]$ which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of $val[]$ such that sum of the weights of this subset is smaller than or equal to W . You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

```
1 def knapSack(W, wt, val, size):
2     k = [[0 for i in range(W+1)] for i in range(size+1)]
3     for i in range(size+1):
4         for w in range(W+1):
5             if i == 0 or w == 0:
6                 k[i][w] = 0
7             elif wt[i-1] <= w:
8                 k[i][w] = max(val[i-1] + k[i-1][w-wt[i-1]], k[i-1][w])
9             else:
10                k[i][w] = k[i-1][w]
11
12     for w in k:
13         print(w)
14
15     return k
16
17
18 # def findElementsInSack(W, matrix, wt, val, size):
19 #     i = size
20 #     row = W
21 #     arr = []
22 #     while i > 0:
23 #         print(matrix[i][row] - matrix[i-1][row - wt[i-1]] )
24 #         print(val[i-1])
25 #         if matrix[i][row] - matrix[i-1][row - wt[i-1]] == val[i-1]:
26 #             arr.append(val[i-1])
27 #             i -= 1
```

```

28 #         row -= wt[i-1]
29 #     else:
30 #         i -= 1
31 #
32 #     return arr
33
34 price = [60, 100, 120]
35 wt = [1, 2, 3]
36 W = 5
37 n = len(price)
38 k = knapSack(W, wt, price, n)
39 print(k[n][W])
40 # print(findElementsInSack(W, k, wt, price, n))

```

Cutting Rod problem

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n .

Determine the maximum value obtainable by cutting up the rod and selling the pieces.

```

1  def cutting_rod(prices, n):
2      dp = [0 for i in range(n+1)]
3      dp[0] = 0
4
5      for i in range(1, n+1):
6          max_val = -float('inf')
7          for j in range(i):
8              max_val = max(max_val, prices[j] + dp[i-j-1])
9          dp[i] = max_val
10
11     return dp[n]
12
13
14 if __name__ == '__main__':
15     arr = [1, 5, 8, 9, 10, 17, 17, 20]
16     size = len(arr)
17     print("Maximum Obtainable Value is " + str(cutting_rod(arr, size)))

```

Minimum number of edits (operations) required to convert 'str1' into 'str2'

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

Insert

Remove

Replace

All of the above operations are of equal cost.

Example:

Input: str1 = "sunday", str2 = "saturday"

Output: 3

Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.

Replace 'n' with 'r', insert t, insert a

```

1  def edit_distance(str1, str2, m, n):
2      matrix = [[0 for i in range(n+1)] for i in range(m+1)]
3
4      for i in range(m+1):
5          for j in range(n+1):
6
7              if i == 0:
8                  matrix[i][j] = j
9
10             elif j == 0:
11                 matrix[i][j] = i
12
13             elif str1[i-1] == str2[j-1]:
14                 matrix[i][j] = matrix[i-1][j-1]
15
16             else:
17                 matrix[i][j] = 1 + min(matrix[i][j-1],           # insert
18                                       matrix[i-1][j],           # remove
19                                       matrix[i-1][j-1])          # replace
20
21         return matrix[m][n]
22
23
24  if __name__ == '__main__':

```

```

25     str1 = 'sunday'
26     str2 = 'saturday'
27
28     print(edit_distance(str1, str2, len(str1), len(str2)))

```

Given a 2-D matrix of 0s and 1s, find the Largest Square which contains all 1s in itself

Given a 2-D matrix of 0s and 1s, find the Largest Square which contains all 1s in itself.

```

1  def find_largest_square(matrix):
2      n = len(matrix)
3
4      # make a matrix for storing the solutions
5      cache = [[0] * n for _ in range(n)]
6      # size of square and its bottom-right indexes
7      size = 0
8      right_indx = -1
9      bottom_indx = -1
10
11     for i in range(n):
12         for j in range(n):
13
14             # if the value is 0 simply move forward as it cannot form a square of 1s
15             if matrix[i][j] == 0:
16                 continue
17
18             # if it is first row or column, copy the matrix values as it is
19             elif i == 0 or j == 0:
20                 cache[i][j] = matrix[i][j]
21
22             # Otherwise, check in the up, left, and diagonally top-left direction fo\
23 r minimum size of square
24             # if all are 1s at these positions in matrix, only then min value will b\
25 e greater than 1
26             # hence add the previous square size to the cache + 1
27             else:
28                 cache[i][j] = 1 + min(cache[i - 1][j], cache[i][j - 1], cache[i - 1]\
29 [j - 1])
30
31             # check if the current square size found is larger than the previously f\

```

```

32  ound size, if so, update it
33      if cache[i][j] > size:
34          size = cache[i][j]
35          bottom_indx, right_indx = i, j
36
37      return size, bottom_indx, right_indx
38
39
40  mat = [[1, 1, 0, 1, 0],
41         [0, 1, 1, 1, 0],
42         [1, 1, 1, 1, 0],
43         [0, 1, 1, 1, 1]]
44  size, bottom, right = find_largest_square(mat)
45
46  if size > 0:
47      print("Size of the square:", size)
48      print("Top-left Co-ordinates:", bottom-size+1, ",", right-size+1)
49      print("Bottom-right Co-ordinates:", bottom, ",", right)
50  else:
51      print("No square of 1s found")

```

Given two sequences, print the longest subsequence present in both of them.

Find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

```

1  def lcs(str1, str2):
2      m = len(str1)
3      n = len(str2)
4
5      matrix = [[0 for i in range(n+1)] for i in range(m+1)]
6
7      for i in range(1, m+1):
8          for j in range(1, n+1):
9
10         if i == 0 or j == 0:

```

```
11         matrix[i][j] = 0
12
13     elif str1[i-1] == str2[j-1]:
14         matrix[i][j] = 1 + matrix[i-1][j-1]
15
16     else:
17         matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])
18
19 index = matrix[m][n]
20
21 res = [""] * index
22 i = m
23 j = n
24
25 while i > 0 and j > 0:
26     if str1[i-1] == str2[j-1]:
27         res[index-1] = str1[i-1]
28         i -= 1
29         j -= 1
30         index -= 1
31
32     elif matrix[i-1][j] > matrix[i][j-1]:
33         i -= 1
34     else:
35         j -= 1
36
37 return res
38
39
40 if __name__ == '__main__':
41     X = "AGGTAB"
42     Y = "GXTXAYB"
43
44     str = ''.join(lcs(X, Y))
45
46     print("Length of longest common subsequence is:", len(str), "\nAnd the subsequenc\
47 e is:", str)
```

Length of the longest subsequence in an array such that all elements of the subsequence are sorted in increasing order

Find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

For example, the length of LIS for is 6 and LIS is.

```

1  def lis(arr):
2      n = len(arr)
3      dp = [1] * n
4
5      for i in range(n):
6          for j in range(i):
7              if arr[j] < arr[i] and dp[j] + 1 > dp[i]:
8                  dp[i] = 1 + dp[j]
9
10     return max(dp)
11
12
13 arr = [10, 22, 9, 33, 21, 50, 41, 60, 80]
14 print(lis(arr))

```

Find minimum cost path in a matrix from (0,0) to given point (m,n)

Given a cost matrix `cost[][]` and a position `(m, n)` in `cost[][]`, write a function that returns cost of minimum cost path to reach `(m, n)` from `(0, 0)`.

Total cost of a path to reach `(m, n)` is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell.

```

1  def min_cost(cost, m, n):
2      dp = [[0 for i in range(n+1)] for i in range(m+1)]
3
4      dp[0][0] = cost[0][0]
5
6      for i in range(1, m+1):
7          dp[i][0] = dp[i-1][0] + cost[i][0]
8
9      for j in range(1, n+1):
10         dp[0][j] = dp[0][j-1] + cost[0][j]
11
12         for i in range(1, m+1):
13             for j in range(1, n+1):
14                 dp[i][j] = cost[i][j] + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
15
16         return dp[m][n]
17
18
19 if __name__ == '__main__':
20     cost = [[1, 2, 3],
21             [4, 8, 2],
22             [1, 5, 3]]
23     m = 2
24     n = 2
25     print("Minimum cost from (0, 0) to ({}, {}) is:".format(m, n), min_cost(cost, m,\
26 n))

```

Partition a set into two subsets such that the difference of subset sums is minimum

```

1  # Partition a set into two subsets such that the difference of subset sums is minimum
2
3
4  def find_min(arr):
5      sum_of_arr = sum(arr)
6      n = len(arr)
7      dp = [[False for i in range(sum_of_arr+1)] for i in range(n+1)]
8
9      for i in range(n+1):
10         dp[i][0] = True
11

```



```

12     for i in range(1, sum_of_arr+1):
13         dp[0][i] = False
14
15     for i in range(1, n+1):
16         for j in range(1, sum_of_arr+1):
17             dp[i][j] = dp[i-1][j]
18
19             if arr[i-1] <= j:
20                 dp[i][j] |= dp[i-1][j - arr[i-1]]
21
22     diff = float('inf')
23
24     for j in range(int(sum_of_arr/2), -1, -1):
25         if dp[n][j] is True:
26             diff = sum_of_arr - 2 * j
27             break
28
29     return diff
30
31
32 if __name__ == '__main__':
33     arr = [3, 1, 4, 2, 2, 1]
34     print("Minimum difference is:", find_min(arr))

```

Minimum number of umbrellas of m different sizes required to accomodate N people

Given N number of people and M different types of unlimited umbrellas. Each m_i in M denotes the exact number of people (m_i), the i th umbrella type can accomodate. Find out the minimum number of umbrellas needed to accomodate all the N people. if such a case is not possible then return -1. Each umbrella has to fill exactly the number of people it can accomodate.

```

1 def min_umbrellas_needed_util(n, umbrellas, dp, count):
2     # if dp has already stored the solution for n people, return
3     if n == 0 or dp[n] != 0:
4         return dp[n]
5
6     min_count = float('inf')
7     curr_count = None
8
9     # Iterate over all the umbrella sizes

```

```
10     for m in umbrellas:
11         # if umbrella can be accomodated fully
12         if n - m > 0:
13             curr_count = min_umbrellas_needed_util(n-m, umbrellas, dp, count+1)
14
15         # if all people are accomodated
16         elif n - m == 0:
17             curr_count = count+1
18
19         # if the umbrella cannot get exact number of people to fit
20         else:
21             curr_count = float('inf')
22
23         if curr_count < min_count:
24             min_count = curr_count
25
26     # memoize result
27     dp[n] = min_count
28
29     return min_count
30
31
32 def min_umbrellas_needed(n, umbrellas):
33     # initialize a dp table for memoization
34     dp = [0] * (n+1)
35
36     count = min_umbrellas_needed_util(n, umbrellas, dp, 0)
37
38     if count == float('inf'):
39         return -1
40
41     return count
42
43 umbrellas = [5,4,2,1]
44 n = 8
45
46 print(f"Number of umbrellas needed to accomodate {n} people: {min_umbrellas_needed(n\
47 , umbrellas)}")
```

Determine if there is a subset of the given set with sum equal to given sum

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

```

1  def isSubsetSum(arr, check_sum):
2      n = len(arr)
3      possible_sum = [[False] * (n + 1) for _ in range(check_sum + 1)]
4
5      for i in range(n+1):
6          possible_sum[0][i] = True
7
8      for i in range(1, check_sum + 1):
9          for j in range(1, n + 1):
10             if i < arr[j - 1]:
11                 possible_sum[i][j] = possible_sum[i][j-1]
12             elif i >= arr[j - 1]:
13                 possible_sum[i][j] = possible_sum[i][j-1] or possible_sum[i - arr[j] \
14 - 1][j-1]
15
16         return possible_sum[check_sum][n]
17
18
19 arr = [3, 34, 4, 12, 5, 2]
20 check_sum = 9
21
22 if isSubsetSum(arr, check_sum):
23     print("Found a subset with sum =", check_sum)
24 else:
25     print("Subset with sum =", check_sum, "Not Found")

```

Given a distance 'dist, count total number of ways to cover the distance with 1, 2 and 3 steps

Given a distance 'dist, count total number of ways to cover the distance with 1, 2 and 3 steps

Input: n = 3

Output: 4

Below are the four ways

1 step + 1 step + 1 step

1 step + 2 step

2 step + 1 step

3 step

```
1 def count_ways(n):
2     count = [0] * (n+1)
3     count[0] = 1
4     count[1] = 1
5     count[2] = 2
6
7     for i in range(3, n+1):
8         count[i] = count[i-1] + count[i-2] + count[i-3]
9
10    return count[n]
11
12
13 if __name__ == '__main__':
14     print(count_ways(4))
```

Chapter 3: Graph

Find all possible words in a board of characters

```
1  # Find all possible words in a board of characters
2  """Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
3      boggle[][] = {{ 'G', 'I', 'Z' },
4                          { 'U', 'E', 'K' },
5                          { 'Q', 'S', 'E' } };
6  Output: Following words of dictionary are present
7      GEEKS
8      QUIZ
9  """
10
11
12 def findWordsUtil(words, boggle, visited, found, r, c, str):
13     rows = len(boggle)
14     cols = len(boggle[0])
15
16     # set the position of character as traversed
17     visited[r][c] = True
18
19     # add the character to string
20     str += boggle[r][c]
21
22     # if the string is in dictionary add it to the set of found words
23     if str in words:
24         found.add(str)
25
26     # traverse all the nearby 8 adjacent cells
27     for i in range(r-1, r+2):
28         for j in range(c-1, c+2):
29             if i >= rows or i < 0 or j >= cols or j < 0 or visited[i][j]:
30                 continue
31             findWordsUtil(words, boggle, visited, found, i, j, str)
32
33     # backtrack and set the status of current character as not traversed
34     visited[r][c] = False
35
```

```

36
37 def findWords(words, boggle):
38     rows = len(boggle)
39     cols = len(boggle[0])
40
41     # initialize a matrix for DFS Traversal
42     visited = [[False for i in range(cols)] for j in range(rows)]
43
44     # set to store the unique found words
45     found = set({})
46     str = ""
47
48     # traverse each character in the boggle and do DFS from there
49     for r in range(rows):
50         for c in range(cols):
51             findWordsUtil(words, boggle, visited, found, r, c, str)
52
53     # return the set of found words
54     return found
55
56 if __name__ == '__main__':
57     words = {"GEEKS", "FOR", "QUIZ", "GO", "SEEK"}
58     boggle = [['G', 'I', 'Z'],
59               ['U', 'E', 'K'],
60               ['Q', 'S', 'E']]
61
62     found = findWords(words, boggle)
63
64     print("Words found in the boggle from the dictionary are:")
65     for word in found:
66         print(word)

```

Breadth First Search Traversal

```
1  # Program to perform breadth first traversal in a graph
2  from collections import defaultdict, deque
3
4
5  class Graph:
6      def __init__(self, directed=False):
7          self.graph = defaultdict(list)
8          self.directed = directed
9
10     def addEdge(self, frm, to):
11         self.graph[frm].append(to)
12
13         if self.directed is False:
14             self.graph[to].append(frm)
15         else:
16             self.graph[to] = self.graph[to]
17
18     def bfsUtil(self, s, visited):
19         queue = deque([])
20         queue.append(s)
21         visited[s] = True
22
23         while queue:
24             vertex = queue.popleft()
25             print(vertex, end=' ')
26
27             # traverse vertices adjacent to vertex
28             for i in self.graph[vertex]:
29                 if not visited[i]:
30                     visited[i] = True
31                     queue.append(i)
32         print()
33
34     def bfs(self, s=None):
35         visited = {i: False for i in self.graph}
36
37         # do bfs from the node specified
38         if s is not None:
39             self.bfsUtil(s, visited)
40         # traverse for all the vertices in other components of graph
41         for v in self.graph:
42             if not visited[v]:
43                 self.bfsUtil(v, visited)
```

```
44
45 if __name__ == '__main__':
46     graph = Graph()
47
48     # component 1 of the graph
49     graph.addEdge(0, 1)
50     graph.addEdge(0, 2)
51     graph.addEdge(1, 2)
52     graph.addEdge(2, 3)
53     graph.addEdge(3, 3)
54     graph.addEdge(1, 4)
55     graph.addEdge(1, 5)
56     graph.addEdge(3, 6)
57
58     # component 2 of the graph
59     graph.addEdge(7, 8)
60     graph.addEdge(8, 9)
61     graph.addEdge(7, 10)
62
63     # call bfs from 2 vertex
64     print("Breadth First Traversal:")
65     graph.bfs(2)
```

Depth First Search Traversal

```
1  # Program to perform depth first traversal in a graph
2  from collections import defaultdict
3
4
5  class Graph:
6      def __init__(self, directed=False):
7          self.graph = defaultdict(list)
8          self.directed = directed
9
10     def addEdge(self, frm, to):
11         self.graph[frm].append(to)
12
13         if self.directed is False:
14             self.graph[to].append(frm)
15         else:
16             self.graph[to] = self.graph[to]
17
```



```
18     def dfsUtil(self, s, visited):
19         stack = []
20         stack.append(s)
21         visited[s] = True
22
23         while stack:
24             vertex = stack.pop()
25             print(vertex, end=' ')
26
27             # traverse vertices adjacent to vertex
28             for i in self.graph[vertex]:
29                 if not visited[i]:
30                     visited[i] = True
31                     stack.append(i)
32         print()
33
34     def dfs(self, s=None):
35         visited = {i: False for i in self.graph}
36
37         # traverse specified vertex
38         if s is not None:
39             self.dfsUtil(s, visited)
40
41         # traverse for all the vertices in other components of graph
42         for v in self.graph:
43             if not visited[v]:
44                 self.dfsUtil(v, visited)
45
46
47 if __name__ == '__main__':
48     # make an undirected graph
49     graph = Graph()
50
51     # component 1 of the graph
52     graph.addEdge(0, 1)
53     graph.addEdge(0, 2)
54     graph.addEdge(1, 2)
55     graph.addEdge(2, 3)
56     graph.addEdge(3, 3)
57     graph.addEdge(1, 4)
58     graph.addEdge(1, 5)
59     graph.addEdge(3, 6)
60
```

```

61     # component 2 of the graph
62     graph.addEdge(7, 8)
63     graph.addEdge(8, 9)
64     graph.addEdge(7, 10)
65
66     # call dfs from 2 vertex
67     print("Depth First Traversal:")
68     graph.dfs(2)

```

Detect Cycle in directed graph

```

1  # Program to detect cycle or loop in a directed graph
2  from collections import defaultdict
3
4
5  class Graph:
6      def __init__(self, directed=False):
7          self.graph = defaultdict(list)
8          self.directed = directed
9
10     def addEdge(self, frm, to):
11         self.graph[frm].append(to)
12
13         if self.directed is False:
14             self.graph[to].append(frm)
15         else:
16             self.graph[to] = self.graph[to]
17
18     def isCyclicUtil(self, s, visited, recurStack):
19
20         if visited[s] is False:
21             recurStack[s] = True
22             visited[s] = True
23
24             # traverse vertices adjacent to vertex
25             for i in self.graph[s]:
26                 if (not visited[i]) and self.isCyclicUtil(i, visited, recurStack):
27                     return True
28                 elif recurStack[i]:
29                     return True
30             recurStack[s] = False
31             return False

```

```

32
33     def isCyclic(self):
34         visited = {i: False for i in self.graph}
35         recurStack = {i: False for i in self.graph}
36
37         # traverse for all the vertices of graph
38         for v in self.graph:
39             if self.isCyclicUtil(v, visited, recurStack):
40                 return True
41         return False
42
43
44 if __name__ == '__main__':
45     # make a directed graph
46     graph = Graph(True)
47
48     graph.addEdge(0, 1)
49     graph.addEdge(0, 2)
50     graph.addEdge(1, 2)
51     graph.addEdge(2, 0)
52     graph.addEdge(2, 3)
53     graph.addEdge(3, 3)
54
55     if graph.isCyclic():
56         print("Cycle exists")
57     else:
58         print("No cycle in the graph")

```

Detect cycle in undirected graph

```

1  # Program to detect cycle or loop in a graph
2  from collections import defaultdict
3
4
5  class Graph:
6      def __init__(self, directed=False):
7          self.graph = defaultdict(list)
8          self.directed = directed
9
10     def addEdge(self, frm, to):
11         # True if edge has been traversed or seen once
12         self.graph[frm].append([to, False])

```

```

13
14     if self.directed is False:
15         self.graph[to].append([frm, False])
16     else:
17         self.graph[to] = self.graph[to]
18
19     def findParent(self, sets, v):
20         if sets[v] == -1:
21             return v
22         else:
23             return self.findParent(sets, sets[v])
24
25     def union(self, sets, x, y):
26         x_set = self.findParent(sets, x)
27         y_set = self.findParent(sets, y)
28         sets[x_set] = y_set
29
30     def isCyclic(self):
31         # sets that show combined vertices or not
32         sets = {i: -1 for i in self.graph}
33
34         for v in self.graph:
35             for e in self.graph[v]:
36                 # if an edge is traversed once skip it
37                 if e[1] is True:
38                     continue
39
40                 # set True for traversing the edge and making union in both adjacenc\
41 y lists         e[1] = True
42
43             for i in self.graph[e[0]]:
44                 if i[0] == v:
45                     i[1] = True
46                     break
47
48                 # find parents of both vertices of the edge
49                 x = self.findParent(sets, v)
50                 y = self.findParent(sets, e[0])
51
52                 # if they share a common parent loop found
53                 if x == y:
54                     return True
55

```

```

56         # union the two vertices in the same set
57         self.union(sets, x, y)
58
59         # if no loop or cycle found return false
60         return False
61
62
63 if __name__ == '__main__':
64     # make a graph
65     graph = Graph()
66     graph.addEdge(0, 1)
67     graph.addEdge(1, 2)
68     graph.addEdge(2, 0)
69
70     if graph.isCyclic():
71         print("Cycle exists in the graph")
72     else:
73         print("No cycle in the graph")

```

Dijkstra's Shortest Path Algorithm

```

1  # Given a graph and a source vertex in graph, find shortest paths from source to all\
2  vertices in the given graph
3  from collections import defaultdict
4
5
6  class Graph:
7      def __init__(self, directed=False):
8          self.graph = defaultdict(list)
9          self.directed = directed
10
11     def addEdge(self, frm, to, weight):
12         self.graph[frm].append([to, weight])
13
14         if self.directed is False:
15             self.graph[to].append([frm, weight])
16         else:
17             self.graph[to] = self.graph[to]
18
19     def find_min(self, dist, visited):
20         minimum = float('inf')
21         index = -1

```

```

22     for v in self.graph.keys():
23         if visited[v] is False and dist[v] < minimum:
24             minimum = dist[v]
25             index = v
26
27     return index
28
29 def dijkstra(self, src):
30     visited = {i: False for i in self.graph}
31     dist = {i: float('inf') for i in self.graph}
32     parent = {i: None for i in self.graph}
33
34     # set distance of src vertex from itself 0
35     dist[src] = 0
36
37     # find shortest path for all vertices
38     for i in range(len(self.graph)-1):
39         # find minimum distance vertex from source
40         # initially src itself as dist[src] = 0
41         u = self.find_min(dist, visited)
42
43         # mark the node as visited
44         visited[u] = True
45         # check if the distance through current edge is less than previously known distance to v
46         for v, w in self.graph[u]:
47
48             if visited[v] is False and dist[u] + w < dist[v]:
49                 dist[v] = dist[u] + w
50                 parent[v] = u
51
52         # return parent list and distance to each node from source
53     return parent, dist
54
55 def printPath(self, parent, v):
56     if parent[v] is None:
57         return
58     self.printPath(parent, parent[v])
59     print(v, end=' ')
60
61 def printSolution(self, dist, parent, src):
62     print('{ }\t{ }\t{ }'.format('Vertex', 'Distance', 'Path'))
63
64     for i in self.graph.keys():

```

```
65         if i == src:
66             continue
67         print('{} -> {} \t \t {} \t \t {}'.format(src, i, dist[i], src), end=' ')
68         self.printPath(parent, i)
69         print()
70
71 if __name__ == '__main__':
72     # make an undirected graph
73     graph = Graph()
74
75     graph.addEdge(0, 1, 4)
76     graph.addEdge(0, 7, 8)
77     graph.addEdge(1, 2, 8)
78     graph.addEdge(1, 7, 11)
79     graph.addEdge(7, 6, 1)
80     graph.addEdge(7, 8, 7)
81     graph.addEdge(6, 8, 6)
82     graph.addEdge(6, 5, 2)
83     graph.addEdge(8, 2, 2)
84     graph.addEdge(2, 3, 7)
85     graph.addEdge(2, 5, 4)
86     graph.addEdge(3, 4, 9)
87     graph.addEdge(3, 5, 14)
88     graph.addEdge(5, 4, 10)
89
90     parent, dist = graph.dijkstra(0)
91
92     graph.printSolution(dist, parent, 0)
```

Find shortest distances between each pair of vertices in a given edge weighted directed Graph

```

1  # Find shortest distances between every pair of vertices in a given edge weighted di\
2  rected Graph
3  """
4
5      10
6      (0)----->(3)
7      |           /\
8      5 |         |
9      |         | 1
10     \\/       |
11     (1)----->(2)
12     3
13 """
14
15 def floyd_warshall(graph):
16     shortest_dist = []
17
18     # copy matrix for storing resultant shortest distances
19     for i in graph:
20         shortest_dist.append(i)
21
22     # Number of vertices in graph
23     V = len(graph) - 1
24
25     # k is intermediate vertex
26     for k in range(V+1):
27         # i is source
28         for i in range(V+1):
29             # j is destination
30             for j in range(V+1):
31                 # store the path which is shorter i.e. min(i->j, i->k->j)
32                 shortest_dist[i][j] = min(shortest_dist[i][j], shortest_dist[i][k] + \
33 shortest_dist[k][j])
34     # return the resultant matrix
35     return shortest_dist
36
37 if __name__ == '__main__':
38     INF = float('inf')
39     graph = [[0, 5, INF, 10],
40              [INF, 0, 3, INF],
41              [INF, INF, 0, 1],
42              [INF, INF, INF, 0]]
43

```



```
44     shortest_dist_matrix = floyd_warshall(graph)
45
46     for i in shortest_dist_matrix:
47         for j in i:
48             if j != float('inf'):
49                 print(j, '\t', end='')
50             else:
51                 print(j, end=' ')
52         print()
```

Graph implementation

```
1  class Vertex:
2      def __init__(self, key):
3          self.key = key
4          self.adjacent = {}
5          self.visited = False
6
7      def setKey(self, key):
8          self.key = key
9
10     def getKey(self):
11         return self.key
12
13     def getVisited(self):
14         return self.visited
15
16     def setVisited(self, val=True):
17         self.visited = val
18
19     def addNeighbour(self, neighbour, weight=0):
20         self.adjacent[neighbour] = weight
21
22     def getNeighbours(self):
23         return self.adjacent.keys()
24
25     def getWeight(self, neighbour):
26         return self.adjacent[neighbour]
27
28
29  class Graph:
30
```

```
31     # Graph is undirected by default
32     def __init__(self, directed=False):
33         self.vertices = {}
34         self.numberOfVertices = 0
35         self.directed = directed
36
37     def addVertex(self, key):
38         node = Vertex(key)
39         self.vertices[key] = node
40         self.numberOfVertices += 1
41         return node
42
43     def addEdge(self, frm, to, weight=0):
44         if frm not in self.vertices:
45             self.addVertex(frm)
46
47         if to not in self.vertices:
48             self.addVertex(to)
49
50         self.vertices[frm].addNeighbour(self.vertices[to], weight)
51
52         if not self.directed:
53             self.vertices[to].addNeighbour(self.vertices[frm], weight)
54
55     def getVertex(self, key):
56         if key in self.vertices:
57             return self.vertices[key]
58         else:
59             return None
60
61     def getVertices(self):
62         return self.vertices.keys()
63
64     def getEdges(self):
65         edges = []
66         for v in self.vertices:
67             edgesFromVertex = []
68
69             for w in self.vertices[v].getNeighbours():
70                 frm = self.vertices[v].getKey()
71                 to = w.getKey()
72                 weight = self.vertices[v].getWeight(w)
73                 edgesFromVertex.append((frm, to, weight))
```

```
74
75         if len(edgesFromVertex) != 0:
76             edges.append(edgesFromVertex)
77
78     return edges
79
80
81 if __name__ == '__main__':
82     g = Graph(directed=False)
83     g.addVertex('a')
84     g.addVertex('b')
85     g.addVertex('c')
86     g.addVertex('d')
87     g.addVertex('e')
88     g.addVertex('f')
89     g.addEdge('a', 'b', 3)
90     g.addEdge('b', 'c', 2)
91     g.addEdge('c', 'd', 1)
92     g.addEdge('d', 'e', 5)
93     g.addEdge('d', 'a', 5)
94     g.addEdge('d', 'a', 2)
95     g.addEdge('e', 'f', 3)
96     g.addEdge('f', 'a', 6)
97     g.addEdge('f', 'b', 6)
98     g.addEdge('f', 'c', 6)
99
100     for edgeSet in g.getEdges():
101         print('edges from', edgeSet[0][0] + ': ', end='')
102         print(edgeSet)
```

Kruskal's Algorithm for Minimum Spanning Tree

```
1  # Kruskal's Minimum Spanning Tree Algorithm
2
3
4  class Graph:
5      def __init__(self, directed=False):
6          self.edges = []
7          self.vertices = set({})
8          self.directed = directed
9
10     def addEdge(self, frm, to, weight):
11         self.edges.append([frm, to, weight])
12         self.vertices.add(frm)
13         self.vertices.add(to)
14
15     def removeEdge(self, frm, to, weight):
16         self.edges.remove([frm, to, weight])
17         flag1 = 0
18         flag2 = 0
19         for f, t, w in self.edges:
20             if frm == f or frm == t:
21                 flag1 = 1
22             if to == f or to == t:
23                 flag2 = 1
24             if flag1 == 1 and flag2 == 1:
25                 break
26
27         if flag1 != 1:
28             self.vertices.remove(frm)
29
30         if flag2 != 1:
31             self.vertices.remove(to)
32
33     def findParent(self, sets, v):
34         if sets[v] == -1:
35             return v
36         else:
37             return self.findParent(sets, sets[v])
38
39     def union(self, sets, x, y):
40         x_set = self.findParent(sets, x)
41         y_set = self.findParent(sets, y)
42         sets[x_set] = y_set
43
```

```
44     def isCyclic(self):
45         # sets that show combined vertices or not
46         sets = {i: -1 for i in self.vertices}
47         for v1, v2, w in self.edges:
48             # find parents of both vertices of the edge
49             x = self.findParent(sets, v1)
50             y = self.findParent(sets, v2)
51
52             # if they share a common parent loop found
53             if x == y:
54                 return True
55             # union the two vertices in the same set
56             self.union(sets, x, y)
57
58             # if no loop or cycle found return false
59             return False
60
61     def kruskalMST(self):
62         g = Graph()
63
64         self.edges = sorted(self.edges, key=lambda x: x[2])
65
66         for frm, to, w in self.edges:
67             if len(g.edges) == len(g.vertices)-1:
68                 break
69             g.addEdge(frm, to, w)
70             if g.isCyclic():
71                 g.removeEdge(frm, to, w)
72         return g
73
74
75 if __name__ == '__main__':
76     # make an undirected graph
77     graph = Graph()
78
79     graph.addEdge(0, 1, 10)
80     graph.addEdge(0, 2, 6)
81     graph.addEdge(0, 3, 5)
82     graph.addEdge(1, 3, 15)
83     graph.addEdge(2, 3, 4)
84
85     new_graph = graph.kruskalMST()
86
```

```
87     for f, t, w in new_graph.edges:
88         print(f, "--", t, "=", w)
```

Topological Sort

```
1  # Program to perform topological sort in a graph
2
3  from collections import defaultdict
4
5
6  class Graph:
7      def __init__(self, directed=False):
8          self.graph = defaultdict(list)
9          self.directed = directed
10
11     def addEdge(self, frm, to):
12         self.graph[frm].append(to)
13
14         if self.directed is False:
15             self.graph[to].append(frm)
16         else:
17             self.graph[to] = self.graph[to]
18
19     def topoSortUtil(self, s, visited, sortList):
20         visited[s] = True
21
22         for i in self.graph[s]:
23             if not visited[i]:
24                 self.topoSortUtil(i, visited, sortList)
25
26         sortList.insert(0, s)
27
28     def topologicalSort(self):
29         visited = {i: False for i in self.graph}
30
31         sortList = []
32         # traverse for all the vertices in other components of graph
33         for v in self.graph:
34             if not visited[v]:
35                 self.topoSortUtil(v, visited, sortList)
36
37         print(sortList)
```

```
38
39
40 if __name__ == '__main__':
41     # make an directed graph
42     g = Graph(directed=True)
43
44     g.addEdge(5, 2)
45     g.addEdge(5, 0)
46     g.addEdge(4, 0)
47     g.addEdge(4, 1)
48     g.addEdge(2, 3)
49     g.addEdge(3, 1)
50
51     # call topologicalSort()
52     print("Topological Sort:")
53     g.topologicalSort()
```

Chapter 4: Heaps

Heap Sort algorithm

Perform Sorting using a max heap in:

$O(n \log n)$ time complexity

$O(1)$ space complexity

```
1  def max_heapify(indx, arr, size):
2      """
3      Assuming sub trees are already max heaps, converts tree rooted at current indx i\
4      nto a max heap.
5      :param indx: Index to check for max heap
6      :param arr: array of elements
7      :param size: size of the array
8      """
9
10     # Get index of left and right child of indx node
11     left_child = indx * 2 + 1
12     right_child = indx * 2 + 2
13
14     largest = indx
15
16     # check what is the largest value node in indx, left child and right child
17     if left_child < size:
18         if arr[left_child] > arr[largest]:
19             largest = left_child
20     if right_child < size:
21         if arr[right_child] > arr[largest]:
22             largest = right_child
23
24     # if indx node is not the largest value, swap with the largest child
25     # and recursively call max_heapify on the respective child swapped with
26     if largest != indx:
27         arr[indx], arr[largest] = arr[largest], arr[indx]
28         max_heapify(largest, arr, size)
29
30
```



```

31 def create_max_heap(arr):
32     """
33     Converts a given array into a max heap
34     :param arr: input array of numbers
35     :return: output max heap
36     """
37     n = len(arr)
38
39     # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
40     # loop from n/2 to 0 index and convert each index node into max heap
41     for i in range(int(n/2), -1, -1):
42         max_heapify(i, arr, n)
43
44
45 def heap_sort(arr):
46     """
47     Sorts the given array using heap sort
48     :param arr: input array to sort
49     """
50
51     create_max_heap(arr)
52     heap_size = len(arr)
53
54     # Swap the max value in heap with the end of the array and decrease the size of \
55 the heap by 1
56     # call max heapify on the 0th index of the array
57     while heap_size > 1:
58         arr[heap_size-1], arr[0] = arr[0], arr[heap_size-1]
59         heap_size -= 1
60         max_heapify(0, arr, heap_size)
61
62
63 heap = [5, 10, 4, 8, 3, 0, 9, 11]
64 heap_sort(heap)
65 print(*heap)

```

Max Heap implementation

A max heap is a complete binary tree [CBT] (implemented using array) in which each node has a value larger than its sub-trees

```

1  from math import ceil
2
3
4  class MaxHeap:
5      def __init__(self, arr=None):
6          self.heap = []
7          self.heap_size = 0
8          if arr is not None:
9              self.create_max_heap(arr)
10             self.heap = arr
11             self.heap_size = len(arr)
12
13     def create_max_heap(self, arr):
14         """
15         Converts a given array into a max heap
16         :param arr: input array of numbers
17         """
18         n = len(arr)
19
20         # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
21         # loop from n/2 to 0 index and convert each index node into max heap
22         for i in range(int(n / 2), -1, -1):
23             self.max_heapify(i, arr, n)
24
25     def max_heapify(self, indx, arr, size):
26         """
27         Assuming sub trees are already max heaps, converts tree rooted at current in\
28 dx into a max heap.
29         :param indx: Index to check for max heap
30         """
31
32         # Get index of left and right child of indx node
33         left_child = indx * 2 + 1
34         right_child = indx * 2 + 2
35
36         largest = indx
37
38         # check what is the largest value node in indx, left child and right child
39         if left_child < size:
40             if arr[left_child] > arr[largest]:
41                 largest = left_child
42         if right_child < size:
43             if arr[right_child] > arr[largest]:

```

```

44         largest = right_child
45
46         # if indx node is not the largest value, swap with the largest child
47         # and recursively call min_heapify on the respective child swapped with
48         if largest != indx:
49             arr[indx], arr[largest] = arr[largest], arr[indx]
50             self.max_heapify(largest, arr, size)
51
52     def insert(self, value):
53         """
54         Inserts an element in the max heap
55         :param value: value to be inserted in the heap
56         """
57         self.heap.append(value)
58         self.heap_size += 1
59
60         indx = self.heap_size - 1
61
62         # Get parent index of the current node
63         parent = int(ceil(indx / 2 - 1))
64
65         # Check if the parent value is smaller than the newly inserted value
66         # if so, then replace the value with the parent value and check with the new\
67     parent
68         while parent >= 0 and self.heap[indx] > self.heap[parent]:
69             self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
70             indx = parent
71             parent = int(ceil(indx / 2 - 1))
72
73     def delete(self, indx):
74         """
75         Deletes the value on the specified index node
76         :param indx: index whose node is to be removed
77         :return: Value of the node deleted from the heap
78         """
79         if self.heap_size == 0:
80             print("Heap Underflow!!")
81             return
82
83         self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
84         self.heap_size -= 1
85
86         self.max_heapify(indx, self.heap, self.heap_size)

```

```

87
88         return self.heap.pop()
89
90     def extract_max(self):
91         """
92         Extracts the maximum value from the heap
93         :return: extracted max value
94         """
95         return self.delete(0)
96
97     def print(self):
98         print(*self.heap)
99
100 heap = MaxHeap([5, 10, 4, 8, 3, 0, 9, 11])
101
102 heap.insert(15)
103 print(heap.delete(2))
104 print(heap.extract_max())
105 heap.print()

```

Min Heap implementation

A Min heap is a complete binary tree [CBT] (implemented using array) in which each node has a value smaller than its sub-trees

```

1  from math import ceil
2
3
4  class MinHeap:
5      def __init__(self, arr=None):
6          self.heap = []
7          self.heap_size = 0
8          if arr is not None:
9              self.create_min_heap(arr)
10             self.heap = arr
11             self.heap_size = len(arr)
12
13     def create_min_heap(self, arr):
14         """
15         Converts a given array into a min heap
16         :param arr: input array of numbers
17         """

```

```

18         n = len(arr)
19
20         # last n/2 elements will be leaf nodes (CBT property) hence already min heaps
21         # loop from n/2 to 0 index and convert each index node into min heap
22         for i in range(int(n / 2), -1, -1):
23             self.min_heapify(i, arr, n)
24
25     def min_heapify(self, indx, arr, size):
26         """
27         Assuming sub trees are already min heaps, converts tree rooted at current in\
28 dx into a min heap.
29         :param indx: Index to check for min heap
30         """
31         # Get index of left and right child of indx node
32         left_child = indx * 2 + 1
33         right_child = indx * 2 + 2
34
35         smallest = indx
36
37         # check what is the smallest value node in indx, left child and right child
38         if left_child < size:
39             if arr[left_child] < arr[smallest]:
40                 smallest = left_child
41         if right_child < size:
42             if arr[right_child] < arr[smallest]:
43                 smallest = right_child
44
45         # if indx node is not the smallest value, swap with the smallest child
46         # and recursively call min_heapify on the respective child swapped with
47         if smallest != indx:
48             arr[indx], arr[smallest] = arr[smallest], arr[indx]
49             self.min_heapify(smallest, arr, size)
50
51     def insert(self, value):
52         """
53         Inserts an element in the min heap
54         :param value: value to be inserted in the heap
55         """
56         self.heap.append(value)
57         self.heap_size += 1
58
59         indx = self.heap_size - 1
60

```

```

61         # Get parent index of the current node
62         parent = int(ceil(indx / 2 - 1))
63
64         # Check if the parent value is smaller than the newly inserted value
65         # if so, then replace the value with the parent value and check with the new\
66     parent
67         while parent >= 0 and self.heap[indx] < self.heap[parent]:
68             self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
69             indx = parent
70             parent = int(ceil(indx / 2 - 1))
71
72     def delete(self, indx):
73         """
74         Deletes the value on the specified index node
75         :param indx: index whose node is to be removed
76         :return: Value of the node deleted from the heap
77         """
78         if self.heap_size == 0:
79             print("Heap Underflow!!")
80             return
81
82         self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
83         self.heap_size -= 1
84
85         self.min_heapify(indx, self.heap, self.heap_size)
86
87         return self.heap.pop()
88
89     def extract_min(self):
90         """
91         Extracts the minimum value from the heap
92         :return: extracted min value
93         """
94         return self.delete(0)
95
96     def print(self):
97         print(*self.heap)
98
99
100 heap = MinHeap([5, 10, 4, 8, 3, 0, 9, 11])
101
102 heap.insert(15)
103 print(heap.delete(2))

```

```
104 print(heap.extract_min())
105 heap.print()
```

Find the median for an incoming stream of numbers after each insertion in the list of numbers

Given an input stream of integers, you must perform the following task for each i -th integer:

1. Add the i -th integer to a running list of integers.
2. Find the median of the updated list (i.e., for the first element through the i -th element).
3. Print the list's updated median on a new line.

- Input Format

The first line contains a single integer, n , denoting the number of integers in the data stream. Each line i of the n subsequent lines contains an integer, a_i , to be added to your list.

- Output Format

After each new integer is added to the list, print the list's updated median on a new line.

Example:

Input:

```
6
12
4
5
3
8
7
```

Output:

```
12.0
8.0
5.0
4.5
5.0
6.0
```

A max heap is a complete binary tree [CBT] (implemented using array) in which each node has a value larger than its sub-trees

```

1  from math import ceil
2
3
4  class MaxHeap:
5      def __init__(self, arr=None):
6          self.heap = []
7          self.heap_size = 0
8          if arr is not None:
9              self.create_max_heap(arr)
10             self.heap = arr
11             self.heap_size = len(arr)
12
13     def create_max_heap(self, arr):
14         """
15         Converts a given array into a max heap
16         :param arr: input array of numbers
17         """
18         n = len(arr)
19
20         # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
21         # loop from n/2 to 0 index and convert each index node into max heap
22         for i in range(int(n / 2), -1, -1):
23             self.max_heapify(i, arr, n)
24
25     def max_heapify(self, indx, arr, size):
26         """
27         Assuming sub trees are already max heaps, converts tree rooted at current in\
28 dx into a max heap.
29         :param indx: Index to check for max heap
30         """
31
32         # Get index of left and right child of indx node
33         left_child = indx * 2 + 1
34         right_child = indx * 2 + 2
35
36         largest = indx
37
38         # check what is the largest value node in indx, left child and right child
39         if left_child < size:
40             if arr[left_child] > arr[largest]:
41                 largest = left_child
42         if right_child < size:
43             if arr[right_child] > arr[largest]:

```



```

44         largest = right_child
45
46         # if indx node is not the largest value, swap with the largest child
47         # and recursively call min_heapify on the respective child swapped with
48         if largest != indx:
49             arr[indx], arr[largest] = arr[largest], arr[indx]
50             self.max_heapify(largest, arr, size)
51
52     def insert(self, value):
53         """
54         Inserts an element in the max heap
55         :param value: value to be inserted in the heap
56         """
57         self.heap.append(value)
58         self.heap_size += 1
59
60         indx = self.heap_size - 1
61
62         # Get parent index of the current node
63         parent = int(ceil(indx / 2 - 1))
64
65         # Check if the parent value is smaller than the newly inserted value
66         # if so, then replace the value with the parent value and check with the new\
67     parent
68         while parent >= 0 and self.heap[indx] > self.heap[parent]:
69             self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
70             indx = parent
71             parent = int(ceil(indx / 2 - 1))
72
73     def delete(self, indx):
74         """
75         Deletes the value on the specified index node
76         :param indx: index whose node is to be removed
77         :return: Value of the node deleted from the heap
78         """
79         if self.heap_size == 0:
80             print("Heap Underflow!!")
81             return
82
83         self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
84         self.heap_size -= 1
85
86         self.max_heapify(indx, self.heap, self.heap_size)

```

```

87
88         return self.heap.pop()
89
90     def extract_max(self):
91         """
92         Extracts the maximum value from the heap
93         :return: extracted max value
94         """
95         return self.delete(0)
96
97     def max(self):
98         return self.heap[0]
99
100
101 class MinHeap:
102     def __init__(self, arr=None):
103         self.heap = []
104         self.heap_size = 0
105         if arr is not None:
106             self.create_min_heap(arr)
107             self.heap = arr
108             self.heap_size = len(arr)
109
110     def create_min_heap(self, arr):
111         """
112         Converts a given array into a min heap
113         :param arr: input array of numbers
114         """
115         n = len(arr)
116
117         # last n/2 elements will be leaf nodes (CBT property) hence already min heaps
118         # loop from n/2 to 0 index and convert each index node into min heap
119         for i in range(int(n / 2), -1, -1):
120             self.min_heapify(i, arr, n)
121
122     def min_heapify(self, indx, arr, size):
123         """
124         Assuming sub trees are already min heaps, converts tree rooted at current in\
125 dx into a min heap.
126         :param indx: Index to check for min heap
127         """
128         # Get index of left and right child of indx node
129         left_child = indx * 2 + 1

```

```

130         right_child = indx * 2 + 2
131
132         smallest = indx
133
134         # check what is the smallest value node in indx, left child and right child
135         if left_child < size:
136             if arr[left_child] < arr[smallest]:
137                 smallest = left_child
138         if right_child < size:
139             if arr[right_child] < arr[smallest]:
140                 smallest = right_child
141
142         # if indx node is not the smallest value, swap with the smallest child
143         # and recursively call min_heapify on the respective child swapped with
144         if smallest != indx:
145             arr[indx], arr[smallest] = arr[smallest], arr[indx]
146             self.min_heapify(smallest, arr, size)
147
148     def insert(self, value):
149         """
150         Inserts an element in the min heap
151         :param value: value to be inserted in the heap
152         """
153         self.heap.append(value)
154         self.heap_size += 1
155
156         indx = self.heap_size - 1
157
158         # Get parent index of the current node
159         parent = int(ceil(indx / 2 - 1))
160
161         # Check if the parent value is smaller than the newly inserted value
162         # if so, then replace the value with the parent value and check with the new\
163 parent
164         while parent >= 0 and self.heap[indx] < self.heap[parent]:
165             self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
166             indx = parent
167             parent = int(ceil(indx / 2 - 1))
168
169     def delete(self, indx):
170         """
171         Deletes the value on the specified index node
172         :param indx: index whose node is to be removed

```

```

173         :return: Value of the node deleted from the heap
174         """
175         if self.heap_size == 0:
176             print("Heap Underflow!!")
177             return
178
179         self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
180         self.heap_size -= 1
181
182         self.min_heapify(indx, self.heap, self.heap_size)
183
184         return self.heap.pop()
185
186     def extract_min(self):
187         """
188         Extracts the minimum value from the heap
189         :return: extracted min value
190         """
191         return self.delete(0)
192
193     def min(self):
194         return self.heap[0]
195
196
197     """
198     Algorithm:
199     *) Split the array stream in 2 halves, min heap(upper array) and max heap (lower arr\
200     ay)
201     *) This way the min and max of the heaps will help you get the median fast.
202     *) Note that the elements should be inserted in the heaps in such a way that the ele\
203     ments
204     in lowerMaxHeap are all smaller than all the elements in the upperMinHeap
205     (i.e., as if the arrays were sorted and then split into two heaps)
206     """
207     n = int(input())
208     upperMinHeap = MinHeap()
209     lowerMaxHeap = MaxHeap()
210
211     for a_i in range(1, n+1):
212         a_t = int(input())
213
214         if lowerMaxHeap.heap_size == 0: # this case occurs only initially when both hea\
215         ps are empty

```

```
216         lowerMaxHeap.insert(a_t)
217     else:
218         # Take example stream 1,2,3,4,9,8,7,6,5 to understand the logic
219         if upperMinHeap.heap_size == lowerMaxHeap.heap_size:
220             if a_t > upperMinHeap.min():
221                 temp = upperMinHeap.extract_min()
222                 lowerMaxHeap.insert(temp)
223                 upperMinHeap.insert(a_t)
224             else:
225                 lowerMaxHeap.insert(a_t)
226         elif a_t > lowerMaxHeap.max():
227             upperMinHeap.insert(a_t)
228         else:
229             temp = lowerMaxHeap.extract_max()
230             upperMinHeap.insert(temp)
231             lowerMaxHeap.insert(a_t)
232
233         # print the median directly if odd number of elements
234         # otherwise average of sum of min heap and max heap tops
235         num = a_i / 2
236         if int(num) != num:
237             print(float(lowerMaxHeap.max()))
238         else:
239             print((lowerMaxHeap.max() + upperMinHeap.min()) / 2)
```

Chapter 5: Linked List

Clone a linked list with next and random pointer

```
1  # Python program to Clone a linked list with next and random pointer
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11         self.random = None
12
13  class SinglyLinkedList:
14
15      # Constructor to initialise head
16      def __init__(self, head=None):
17          self.head = None
18
19      # Function to create copy the linked list
20      def copy_list(self, l2):
21          # if first list is empty return
22          if self.head is None:
23              return
24
25          curr1 = self.head
26
27          # insert new nodes with same data as linst at adjacent positions of each node
28          while curr1 is not None:
29              node = Node(curr1.data)
30              temp = curr1.next
31              curr1.next = node
32              node.next = temp
33              curr1 = temp
34
35          curr1 = self.head
```

```

36         # update the random pointers of even node to point to the random nodes of odd
37 d nodes next
38         while curr1 is not None:
39             if curr1.random is not None:
40                 curr1.next.random = curr1.random.next
41                 curr1 = curr1.next.next
42
43         curr1 = self.head
44         curr2 = l2.head
45         # assign even nodes to the Copy linked list to make the new list
46         # re-assign the old links of list 1
47         while curr1 is not None:
48             if l2.head is None:
49                 l2.head = curr1.next
50                 curr2 = l2.head
51             else:
52                 curr2.next = curr1.next
53                 curr2 = curr2.next
54                 curr1.next = curr1.next.next
55                 curr2.next = None
56                 curr1 = curr1.next
57
58         # Function to Insert data at the beginning of the linked list
59         def insert_at_beg(self, data):
60             node = Node(data)
61             node.next = self.head
62             self.head = node
63
64         # Function to print the linked list
65         def print_data(self):
66             current = self.head
67             # print data of each node
68             while current is not None:
69                 print(current.data, '-> ', end='')
70                 current = current.next
71
72             current = self.head
73             print('None')
74             while current is not None:
75                 print("V", ' ', end='')
76                 current = current.next
77
78             current = self.head

```

```
79         print()
80         # print random pointer node's data of each node
81         while current is not None:
82             if current.random is not None:
83                 print(current.random.data, ' ', end='')
84             else:
85                 print("N", ' ', end='')
86             current = current.next
87         print()
88
89 if __name__ == '__main__':
90     linked_list = SinglyLinkedList()
91     # make nodes
92     node1 = Node(1)
93     node2 = Node(2)
94     node3 = Node(3)
95     node4 = Node(4)
96     node5 = Node(5)
97     # set next pointer of each node
98     node1.next = node2
99     node2.next = node3
100    node3.next = node4
101    node4.next = node5
102    # set random pointer of each node
103    node1.random = node3
104    node2.random = node1
105    node3.random = node5
106    node4.random = node3
107    node5.random = node2
108    # assing the nodes list to head
109    linked_list.head = node1
110
111    # print the data of linked list
112    print('original list:')
113    linked_list.print_data()
114
115    # make empty LL2 to copy data of LL1 in
116    linked_list2 = SinglyLinkedList(Node())
117
118    # copy LL1 into LL2
119    linked_list.copy_list(linked_list2)
120
121    # print the copied linked list
```



```

122     print('copied list:')
123     linked_list2.print_data()

```

Given a linked list of line segments, remove middle points if any

Given a linked list of co-ordinates where adjacent points either form a vertical line or a horizontal line.

Delete points from the linked list which are in the middle of a horizontal or vertical line.

Input: (0,10)->(1,10)->(5,10)->(7,10)

|

(7,5)->(20,5)->(40,5)

Output: Linked List should be changed to following

(0,10)->(7,10)

|

(7,5)->(40,5)

```

1  # Node class
2  class Node:
3
4      # Constructor to initialise (x, y) coordinates and next
5      def __init__(self, x=None, y=None):
6          self.x = x
7          self.y = y
8          self.next = None
9
10
11 class SinglyLinkedList:
12
13     # Constructor to initialise head
14     def __init__(self):
15         self.head = None
16
17     # Function to find middle node of a linked list
18     def delete_middle_nodes(self):
19         current = self.head
20
21         # iterate while the next of the next node is not none
22         while current and current.next and current.next.next is not None:

```

```

23         # assign variables for next and next of next nodes
24         next = current.next
25         next_next = current.next.next
26
27         # if x coordinates are equal of current and next node i.e. horizontal li\
28 ne
29         if current.x == next.x:
30             # check if there are more than 2 nodes in the horizontal line
31             # if yes then delete the middle node and update next and next_next
32             while next_next is not None and next.x == next_next.x:
33                 current.next = next_next
34                 next = next_next
35                 next_next = next_next.next
36         # if y coordinates are equal of current and next node i.e. vertical line
37         elif current.y == next.y:
38             # check if there are more than 2 nodes in the vertical line
39             # if yes then delete the middle node and update next and next_next
40             while next_next is not None and next.y == next_next.y:
41                 current.next = next_next
42                 next = next_next
43                 next_next = next_next.next
44         # updated the current node to next node for checking the next line nodes
45         current = current.next
46
47     # Function to Insert data at the beginning of the linked list
48     def insert_at_beg(self, x, y):
49         node = Node(x, y)
50         node.next = self.head
51         self.head = node
52
53     # Function to print the linked list
54     def print_data(self):
55         current = self.head
56         while current is not None:
57             print('(', current.x, ', ', current.y, ') -> ', end='')
58             current = current.next
59         print('None')
60
61 if __name__ == '__main__':
62     linked_list = SinglyLinkedList()
63     linked_list.insert_at_beg(40,5)
64     linked_list.insert_at_beg(20,5)
65     linked_list.insert_at_beg(7,5)

```

```

66     linked_list.insert_at_beg(7,10)
67     linked_list.insert_at_beg(5,10)
68     linked_list.insert_at_beg(1,10)
69     linked_list.insert_at_beg(0,10)
70
71     # print the linked list representing vertical and horizontal lines
72     linked_list.print_data()
73
74     # call the delete_middle_nodes function
75     linked_list.delete_middle_nodes()
76
77     # print the new linked list
78     linked_list.print_data()

```

Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes.

When constructing the result list, we may switch to the other input list only at the point of intersection (which mean the two node with the same value in the lists). You are allowed to use $O(1)$ extra space.

```

1  # Python program to Construct a Maximum Sum Linked List out of two Sorted Linked Lis\
2  ts having some Common nodes.
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to find max sum linked list using 2 linked lists
20     def max_sum_list(self, l2):

```

```

21     pre1 = cur1 = self.head
22     pre2 = cur2 = l2.head
23     result = None
24
25     # while any of the lists are not empty keep traversing
26     while cur1 or cur2 is not None:
27         sum1 = sum2 = 0
28         while cur1 and cur2 is not None and cur1.data != cur2.data:
29
30             # if 1st list's node data is less than 2nd list's node
31             if cur1.data < cur2.data:
32                 sum1 += cur1.data
33                 cur1 = cur1.next
34
35             # if 2nd list's node data is less than 1st list's node
36             else:
37                 sum2 += cur2.data
38                 cur2 = cur2.next
39
40         # if any of the list has ended calculate the sum of the other list till \
41 the end
42         if cur1 is None:
43             while cur2 is not None:
44                 sum2 += cur2.data
45                 cur2 = cur2.next
46         elif cur2 is None:
47             while cur1 is not None:
48                 sum1 += cur1.data
49                 cur1 = cur1.next
50
51         # initial case when result's head needs to be set
52         if pre1 is self.head and pre2 is l2.head:
53             result = pre1 if sum1 > sum2 else pre2
54         else:
55             if sum1 > sum2:
56                 pre2.next = pre1.next
57             else:
58                 pre1.next = pre2.next
59
60     pre1 = cur1
61     pre2 = cur2
62
63     if cur1 is not None:

```

```
64         cur1 = cur1.next
65         if cur2 is not None:
66             cur2 = cur2.next
67
68     return result
69
70     # Function to Insert data at the beginning of the linked list
71     def insert_at_beg(self, data):
72         node = Node(data)
73         node.next = self.head
74         self.head = node
75
76     # Function to print the linked list
77     def print_data(self):
78         current = self.head
79         while current is not None:
80             print(current.data, '-> ', end='')
81             current = current.next
82         print('None')
83
84 if __name__ == '__main__':
85     linked_list1 = SinglyLinkedList()
86     linked_list1.insert_at_beg(130)
87     linked_list1.insert_at_beg(120)
88     linked_list1.insert_at_beg(100)
89     linked_list1.insert_at_beg(90)
90     linked_list1.insert_at_beg(32)
91     linked_list1.insert_at_beg(12)
92     linked_list1.insert_at_beg(3)
93     linked_list1.insert_at_beg(0)
94
95     linked_list2 = SinglyLinkedList()
96     linked_list2.insert_at_beg(120)
97     linked_list2.insert_at_beg(110)
98     linked_list2.insert_at_beg(90)
99     linked_list2.insert_at_beg(30)
100    linked_list2.insert_at_beg(3)
101    linked_list2.insert_at_beg(1)
102
103    print('List 1:')
104    linked_list1.print_data()
105    print('List 2:')
106    linked_list2.print_data()
```

```

107
108     # call the max_sum_list function and update the list 1's head to point the max s\
109 um list
110     linked_list1.head = linked_list1.max_sum_list(linked_list2)
111
112     # print the max sum linked list
113     print('Max sum linked list:')
114     linked_list1.print_data()

```

Merge a linked list into another linked list at alternate positions

```

1  # Python program to Merge a linked list into another linked list at alternate positi\
2  ons
3
4
5  # Node class
6  class Node:
7
8      # Constructor to initialise data and next
9      def __init__(self, data=None):
10         self.data = data
11         self.next = None
12
13
14  class SinglyLinkedList:
15
16      # Constructor to initialise head
17      def __init__(self):
18         self.head = None
19
20      # Function to merge 2 linked lists at alternate positions
21      def merge(self, l2):
22         h1 = self.head
23         h2 = l2.head
24
25         # Merge at alternate positions until the h1 has alternate positions
26         while h1 and h2 is not None:
27             h1_next = h1.next
28             h2_next = h2.next
29

```

```

30         h1.next = h2
31         h2.next = h1_next
32
33         h1 = h1_next
34         h2 = h2_next
35
36         # update the head of h2 if linked list remains i.e. no more alternate positi\
37     ons in h1
38         l2.head = h2
39
40     # Function to Insert data at the beginning of the linked list
41     def insert_at_beg(self, data):
42         node = Node(data)
43         node.next = self.head
44         self.head = node
45
46     # Function to print the linked list
47     def print_data(self):
48         current = self.head
49         while current is not None:
50             print(current.data, '-> ', end='')
51             current = current.next
52         print('None')
53
54     if __name__ == '__main__':
55         linked_list1 = SinglyLinkedList()
56         linked_list1.insert_at_beg(9)
57         linked_list1.insert_at_beg(8)
58         linked_list1.insert_at_beg(7)
59         linked_list1.insert_at_beg(6)
60         linked_list1.insert_at_beg(5)
61
62         linked_list2 = SinglyLinkedList()
63         linked_list2.insert_at_beg(12)
64         linked_list2.insert_at_beg(11)
65         linked_list2.insert_at_beg(10)
66         linked_list2.insert_at_beg(4)
67         linked_list2.insert_at_beg(3)
68         linked_list2.insert_at_beg(2)
69         linked_list2.insert_at_beg(1)
70
71         print('List 1:')
72         linked_list1.print_data()

```

```

73     print('List 2:')
74     linked_list2.print_data()
75
76     # call the merge function
77     linked_list1.merge(linked_list2)
78
79     # print the merged linked list
80     print('Merged list:')
81     linked_list1.print_data()
82     linked_list2.print_data()

```

Perform Merge Sort

```

1  # Python program to perform Merge Sort on a Singly linked list
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self, head=None):
17         self.head = head
18
19     # Function to Insert data at the beginning of the linked list
20     def insert_at_beg(self, data):
21         node = Node(data)
22         node.next = self.head
23         self.head = node
24
25     # Function to print the linked list
26     def print_data(self):
27         current = self.head
28         while current is not None:
29             print(current.data, '-> ', end='')

```



```
30         current = current.next
31         print('None')
32
33
34 # Function to split the linked list into two halves
35 def split(head):
36     slow = head
37
38     if slow is None or slow.next is None:
39         return head, None
40
41     fast = slow.next
42
43     # Reach to the middle of the linked list
44     while fast is not None:
45         fast = fast.next
46         if fast is not None:
47             fast = fast.next
48             slow = slow.next
49
50     fast = slow.next
51     # break the linked list in half
52     slow.next = None
53
54     # return the 2 linked lists formed
55     return head, fast
56
57
58 # Function to merge linked lists in sorted order
59 def merge(a, b):
60     # Make a dummy node
61     dummy = Node()
62     # dummy node next will be the head of our merged list
63     dummy.next = None
64
65     temp = SinglyLinkedList(dummy)
66     tail = temp.head
67
68     while True:
69         if a is None:
70             tail.next = b
71             break
72         elif b is None:
```

```
73         tail.next = a
74         break
75     elif a.data <= b.data:
76         tail.next = a
77         a = a.next
78     else:
79         tail.next = b
80         b = b.next
81     tail = tail.next
82
83     return temp.head.next
84
85
86 # Function Merge Sort
87 def merge_sort(head):
88
89     if head is None or head.next is None:
90         return head
91
92     a, b = split(head)
93     a = merge_sort(a)
94     b = merge_sort(b)
95
96     head = merge(a, b)
97
98     return head
99
100
101 if __name__ == '__main__':
102     linked_list = SinglyLinkedList()
103     linked_list.insert_at_beg(9)
104     linked_list.insert_at_beg(3)
105     linked_list.insert_at_beg(2)
106     linked_list.insert_at_beg(1)
107     linked_list.insert_at_beg(5)
108     linked_list.insert_at_beg(4)
109     linked_list.insert_at_beg(8)
110     linked_list.insert_at_beg(7)
111     linked_list.insert_at_beg(6)
112
113     # before sorting
114     print('before sorting')
115     linked_list.print_data()
```

```
116
117     # call merge_sort function
118     linked_list.head = merge_sort(linked_list.head)
119
120     # after sorting
121     print('after sorting')
122     linked_list.print_data()
```

Find Middle Node

```
1  # Python program to find middle node of a singly linked list
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to find middle node of a linked list
20     def find_mid(self):
21         fast = self.head
22         slow = self.head
23
24         # Make fast run twice the speed of slow
25         # when fast is at the last of the list
26         # slow will be at the mid node
27         while fast is not None:
28             fast = fast.next
29             if fast is not None:
30                 fast = fast.next
31                 slow = slow.next
32
33         return slow
```

```
34
35     # Function to Insert data at the beginning of the linked list
36     def insert_at_beg(self, data):
37         node = Node(data)
38         node.next = self.head
39         self.head = node
40
41     # Function to print the linked list
42     def print_data(self):
43         current = self.head
44         while current is not None:
45             print(current.data, '-> ', end='')
46             current = current.next
47         print('None')
48
49 if __name__ == '__main__':
50     linked_list = SinglyLinkedList()
51     linked_list.insert_at_beg(9)
52     linked_list.insert_at_beg(8)
53     linked_list.insert_at_beg(7)
54     linked_list.insert_at_beg(6)
55     linked_list.insert_at_beg(5)
56     linked_list.insert_at_beg(4)
57     linked_list.insert_at_beg(3)
58     linked_list.insert_at_beg(2)
59     linked_list.insert_at_beg(1)
60     linked_list.print_data()
61     # call the find_mid function
62     mid = linked_list.find_mid()
63     # print the middle node if not None
64     if mid is not None:
65         print(mid.data)
```

Point to next higher value node in a linked list with an arbitrary pointer

```
1  # Python program to Point to next higher value node in a linked list with an arbitra\
2  ry pointer
3
4
5  # Node class
6  class Node:
7
8      # Constructor to initialise data and next and arbitrary pointer
9      def __init__(self, data=None):
10         self.data = data
11         self.next = None
12         self.arbit = None
13
14
15  class SinglyLinkedList:
16
17      # Constructor to initialise head
18      def __init__(self, head=None):
19         self.head = head
20
21      # Function to Insert data at the beginning of the linked list
22      def insert_at_beg(self, data):
23         node = Node(data)
24         node.next = self.head
25         node.arbit = node.next
26         self.head = node
27
28      # Function to print the linked list
29      def print_data(self):
30         current = self.head
31         # print data of each node
32         while current is not None:
33             print(current.data, '-> ', end='')
34             current = current.next
35
36         current = self.head
37         print('None')
38         while current is not None:
39             print("V", ' ', end='')
40             current = current.next
41
42         current = self.head
43         print()
```

```
44         # print arbitrary pointer node's data of each node
45         while current is not None:
46             if current.arbit is not None:
47                 print(current.arbit.data, ' ', end='')
48             else:
49                 print("N", ' ', end='')
50             current = current.next
51         print()
52
53
54     # Function to split the linked list into two halves
55     def split(head):
56         slow = head
57
58         if slow is None or slow.arbit is None:
59             return head, None
60
61         fast = slow.arbit
62
63         # Reach to the middle of the linked list
64         while fast is not None:
65             fast = fast.arbit
66             if fast is not None:
67                 fast = fast.arbit
68                 slow = slow.arbit
69
70         fast = slow.arbit
71         # break the linked list in half
72         slow.arbit = None
73
74         # return the 2 linked lists formed
75         return head, fast
76
77
78     # Function to merge linked lists in sorted order
79     def merge(a, b):
80         # Make a dummy node
81         dummy = Node()
82         # dummy node arbit will be the head of our merged list
83         dummy.arbit = None
84
85         temp = SinglyLinkedList(dummy)
86         tail = temp.head
```

```
87
88     while True:
89         if a is None:
90             tail.arbit = b
91             break
92         elif b is None:
93             tail.arbit = a
94             break
95         elif a.data <= b.data:
96             tail.arbit = a
97             a = a.arbit
98         else:
99             tail.arbit = b
100            b = b.arbit
101            tail = tail.arbit
102
103     return temp.head.arbit
104
105
106 # Function Merge Sort
107 def merge_sort(head):
108
109     if head is None or head.arbit is None:
110         return head
111
112     a, b = split(head)
113     a = merge_sort(a)
114     b = merge_sort(b)
115
116     head = merge(a, b)
117
118     return head
119
120
121 if __name__ == '__main__':
122     linked_list = SinglyLinkedList()
123     linked_list.insert_at_beg(3)
124     linked_list.insert_at_beg(2)
125     linked_list.insert_at_beg(10)
126     linked_list.insert_at_beg(5)
127
128     # before linking the arbit
129     print('before linking')
```

```
130     linked_list.print_data()
131
132     # call merge_sort function
133     # to sort the linked list on the basis of the arbitrary pointers
134     merge_sort(linked_list.head)
135
136     # after linking the arbit
137     print('after linking')
138     linked_list.print_data()
```

Find if linked list contains any cycle

```
1  # Python program to check if the singly linked list contains cycle or not
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13  class SinglyLinkedList:
14
15      # Constructor to initialise head
16      def __init__(self):
17          self.head = None
18
19      # Function to find cycle in linked list
20      def find_cycle(self):
21          fast = self.head.next
22          slow = self.head
23
24          # Make fast run twice the speed of slow
25          # if fast coincide with slow
26          # then there is a loop or cycle
27          while fast is not None:
28              # return True if cycle
29              if fast is slow:
30                  return True
```



```
31         fast = fast.next
32         if fast is not None:
33             fast = fast.next
34             slow = slow.next
35
36         # return False if no cycle
37         return False
38
39         # Function to Insert data at the beginning of the linked list
40     def insert_at_beg(self, data):
41         node = Node(data)
42         node.next = self.head
43         self.head = node
44
45         # Function to print the linked list
46     def print_data(self):
47         current = self.head
48         while current is not None:
49             print(current.data, '-> ', end='')
50             current = current.next
51             print('None')
52
53 if __name__ == '__main__':
54     linked_list = SinglyLinkedList()
55     linked_list.insert_at_beg(9)
56     linked_list.insert_at_beg(8)
57     linked_list.insert_at_beg(7)
58     linked_list.insert_at_beg(6)
59     linked_list.insert_at_beg(5)
60     linked_list.insert_at_beg(4)
61     linked_list.insert_at_beg(3)
62     linked_list.insert_at_beg(2)
63     linked_list.insert_at_beg(1)
64
65     temp = head = linked_list.head
66
67     # get pointer to the end of the list
68     while temp.next is not None:
69         temp = temp.next
70
71     # Make a loop in the list
72     temp.next = head.next.next.next
73
```

```
74     # call the find_cycle function
75     result = linked_list.find_cycle()
76
77     # print if cycle or not
78     print('Yes! there is a cycle') if result else print('No! there is no cycle')
```

To select a random node in a singly linked list

```
1  # to select a random node in a singly linked list
2  import random
3
4
5  # Node class
6  class Node:
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to get random node in linked list
20     def get_random_node(self):
21         if self.head is None:
22             return None
23
24         random.seed()
25         random_node = self.head
26         current = self.head.next
27         n = 2
28         while current is not None:
29             if random.randrange(n) == 0:
30                 random_node = current
31                 current = current.next
32                 n += 1
33         return random_node
34
```

```
35     # Function to Insert data at the beginning of the linked list
36     def insert_at_beg(self, data):
37         node = Node(data)
38         node.next = self.head
39         self.head = node
40
41     # Function to print the linked list
42     def print_data(self):
43         current = self.head
44         while current is not None:
45             print(current.data, '-> ', end='')
46             current = current.next
47         print('None')
48
49 if __name__ == '__main__':
50     linked_list = SinglyLinkedList()
51     linked_list.insert_at_beg(9)
52     linked_list.insert_at_beg(8)
53     linked_list.insert_at_beg(7)
54     linked_list.insert_at_beg(6)
55     linked_list.insert_at_beg(5)
56     linked_list.insert_at_beg(4)
57     linked_list.insert_at_beg(3)
58     linked_list.insert_at_beg(2)
59     linked_list.insert_at_beg(1)
60
61     random_node = linked_list.get_random_node()
62     print("Random node data is:")
63     print(random_node.data)
```

Find and remove cycle if any

```
1  # Python program to check if the singly linked list contains cycle or not
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13  class SinglyLinkedList:
14
15      # Constructor to initialise head
16      def __init__(self):
17          self.head = None
18
19      # Function to find cycle in linked list
20      def find_cycle_remove(self):
21          fast = self.head.next
22          slow = self.head
23
24          # Make fast run twice the speed of slow
25          # if fast coincide with slow
26          # then there is a loop or cycle
27          while fast is not None:
28              # break loop if cycle exists
29              if fast is slow:
30                  break
31              fast = fast.next
32              if fast is not None:
33                  fast = fast.next
34                  slow = slow.next
35
36          if fast is slow:
37              slow = self.head
38              while fast.next is not slow:
39                  fast = fast.next
40                  slow = slow.next
41              fast.next = None
42              return True
43
```

```
44         # return False if no cycle
45         return False
46
47     # Function to Insert data at the beginning of the linked list
48     def insert_at_beg(self, data):
49         node = Node(data)
50         node.next = self.head
51         self.head = node
52
53     # Function to print the linked list
54     def print_data(self):
55         current = self.head
56         while current is not None:
57             print(current.data, '-> ', end='')
58             current = current.next
59         print('None')
60
61     if __name__ == '__main__':
62         linked_list = SinglyLinkedList()
63         linked_list.insert_at_beg(9)
64         linked_list.insert_at_beg(8)
65         linked_list.insert_at_beg(7)
66         linked_list.insert_at_beg(6)
67         linked_list.insert_at_beg(5)
68         linked_list.insert_at_beg(4)
69         linked_list.insert_at_beg(3)
70         linked_list.insert_at_beg(2)
71         linked_list.insert_at_beg(1)
72
73         temp = head = linked_list.head
74
75         # get pointer to the end of the list
76         while temp.next is not None:
77             temp = temp.next
78
79         # Make a loop in the list
80         temp.next = head.next.next.next
81
82         # call the find_cycle function
83         result = linked_list.find_cycle_remove()
84
85         # print if cycle or not
86         print('Yes! there was a cycle') if result else print('No! there was no cycle')
```

```
87     linked_list.print_data()

## Reverse sub list of K nodes each

1  # Python program to reverse a linked list in group of given size k
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to reverse K nodes of linked list
20     def reverse_k_nodes(self, head, k):
21         current = head
22         next = None
23         prev = None
24         count = 0
25
26         # traverse k nodes ahead reversing the links or until current is not None
27         while current is not None and count < k:
28             next = current.next
29             current.next = prev
30             prev = current
31             current = next
32             count += 1
33
34         # recursive call to the function to reverse the remaining n-k nodes
35         if next is not None:
36             head.next = self.reverse_k_nodes(next, k)
37
38         # return the new header of the current sublist
39         return prev
```

```
40
41     # Function to Insert data at the beginning of the linked list
42     def insert_at_beg(self, data):
43         node = Node(data)
44         node.next = self.head
45         self.head = node
46
47     # Function to print the linked list
48     def print_data(self):
49         current = self.head
50         while current is not None:
51             print(current.data, '-> ', end='')
52             current = current.next
53         print('None')
54
55 if __name__ == '__main__':
56     linked_list = SinglyLinkedList()
57     linked_list.insert_at_beg(7)
58     linked_list.insert_at_beg(6)
59     linked_list.insert_at_beg(5)
60     linked_list.insert_at_beg(4)
61     linked_list.insert_at_beg(3)
62     linked_list.insert_at_beg(2)
63     linked_list.insert_at_beg(1)
64     linked_list.print_data()
65     # call the reverse k nodes function
66     linked_list.head = linked_list.reverse_k_nodes(linked_list.head, 3)
67     # print the reversed list
68     linked_list.print_data()
```

Reverse alternate sub list of K nodes each

```
1  # Python program to alternately reverse a linked list in group of given size k
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to reverse K nodes of linked list
20     def reverse_k_nodes(self, head, k):
21         current = head
22         next = None
23         prev = None
24         count = 0
25
26         # traverse k nodes ahead reversing the links or until current is not None
27         while current is not None and count < k:
28             next = current.next
29             current.next = prev
30             prev = current
31             current = next
32             count += 1
33
34         if head is not None:
35             head.next = current
36
37         count = 0
38         # traverse the k nodes to be skipped
39         while current is not None and count < k-1:
40             current = current.next
41             count += 1
42
43         # recursive call to the function to alternate reverse the remaining n-2k nodes
```



```

44 es
45     if current is not None:
46         current.next = self.reverse_k_nodes(current.next, k)
47
48     # return the new header of the current sublist
49     return prev
50
51     # Function to Insert data at the beginning of the linked list
52     def insert_at_beg(self, data):
53         node = Node(data)
54         node.next = self.head
55         self.head = node
56
57     # Function to print the linked list
58     def print_data(self):
59         current = self.head
60         while current is not None:
61             print(current.data, '-> ', end='')
62             current = current.next
63         print('None')
64
65 if __name__ == '__main__':
66     linked_list = SinglyLinkedList()
67     linked_list.insert_at_beg(10)
68     linked_list.insert_at_beg(9)
69     linked_list.insert_at_beg(8)
70     linked_list.insert_at_beg(7)
71     linked_list.insert_at_beg(6)
72     linked_list.insert_at_beg(5)
73     linked_list.insert_at_beg(4)
74     linked_list.insert_at_beg(3)
75     linked_list.insert_at_beg(2)
76     linked_list.insert_at_beg(1)
77     linked_list.print_data()
78     # call the reverse k nodes function
79     linked_list.head = linked_list.reverse_k_nodes(linked_list.head, 3)
80     # print the reversed list
81     linked_list.print_data()

```

Reverse a linked list

```
1  # Python program to reverse a singly linked list
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head
16     def __init__(self):
17         self.head = None
18
19     # Function to reverse a linked list
20     def reverse(self):
21
22         # If linked list is empty
23         if self.head is None:
24             return None
25
26         current = self.head
27         prev = None
28
29         while current is not None:
30             # Store the value of current.next
31             next = current.next
32             # Set current.next to point to the previous node
33             current.next = prev
34             # Update pointers for next iteration
35             prev = current
36             current = next
37
38         self.head = prev
39
40     # Function to Insert data at the beginning of the linked list
41     def insert_at_beg(self, data):
42         node = Node(data)
43         node.next = self.head
```

```

44         self.head = node
45
46     # Function to print the linked list
47     def print_data(self):
48         current = self.head
49         while current is not None:
50             print(current.data, '-> ', end='')
51             current = current.next
52         print('None')
53
54 if __name__ == '__main__':
55     linked_list = SinglyLinkedList()
56     linked_list.insert_at_beg(7)
57     linked_list.insert_at_beg(6)
58     linked_list.insert_at_beg(5)
59     linked_list.insert_at_beg(4)
60     linked_list.insert_at_beg(3)
61     linked_list.insert_at_beg(2)
62     linked_list.insert_at_beg(1)
63     linked_list.print_data()
64     # call the reverse function
65     linked_list.reverse()
66     # print the reversed list
67     linked_list.print_data()

```

Bring even valued nodes to the front

```

1  # Python program to segregate Even and Odd value nodes in a singly linked list
2
3
4  # Node class
5  class Node:
6
7      # Constructor to initialise data and next
8      def __init__(self, data=None):
9          self.data = data
10         self.next = None
11
12
13 class SinglyLinkedList:
14
15     # Constructor to initialise head

```

```
16     def __init__(self):
17         self.head = None
18
19         # Function to segregate even odd value nodes of linked list
20     def segregateEvenOdd(self):
21         current = None
22         prev = self.head
23         pivot = None
24
25         # If empty list or single element in the list
26         if self.head is None or self.head.next is None:
27             return self.head
28
29         # if the first node is even
30         # initialise pivot as head
31         if prev.data % 2 == 0:
32             pivot = self.head
33             current = prev.next
34         # else find the first node in the list that is even
35         # make that node the head of the list
36         # initialise pivot as head
37         else:
38             while prev.next is not None:
39                 if prev.next.data % 2 == 0:
40                     pivot = prev.next
41                     prev.next = pivot.next
42                     pivot.next = self.head
43                     self.head = pivot
44                     current = prev.next
45                     break
46             prev = prev.next
47
48         # keep moving the current pointer and prev pointer
49         while current is not None:
50             # if even value is found at the node
51             if current.data % 2 == 0:
52                 # if the node is adjacent to pivot
53                 # simply increment the pivot to next node
54                 # and shift prev and current one step ahead
55                 if prev is pivot:
56                     pivot = current
57                     prev = current
58                     current = current.next
```

```

59         # else insert the node after the pivot
60         # shift the pivot to the newly inserted node
61         # update current and prev
62         else:
63             prev.next = current.next
64             current.next = pivot.next
65             pivot.next = current
66             pivot = current
67             current = prev.next
68         # if odd value simply increment current and prev
69         else:
70             prev = current
71             current = current.next
72
73         # return the updated linked list head
74         return self.head
75
76     # Function to Insert data at the beginning of the linked list
77     def insert_at_beg(self, data):
78         node = Node(data)
79         node.next = self.head
80         self.head = node
81
82     # Function to print the linked list
83     def print_data(self):
84         current = self.head
85         while current is not None:
86             print(current.data, '-> ', end='')
87             current = current.next
88         print('None')
89
90     if __name__ == '__main__':
91         linked_list = SinglyLinkedList()
92         linked_list.insert_at_beg(7)
93         linked_list.insert_at_beg(6)
94         linked_list.insert_at_beg(5)
95         linked_list.insert_at_beg(4)
96         linked_list.insert_at_beg(2)
97         linked_list.insert_at_beg(3)
98         linked_list.insert_at_beg(2)
99         print('Before segregation:', end=' ')
100        linked_list.print_data()
101        linked_list.head = linked_list.segregateEvenOdd()

```

```

102     print('After segregation:', end=' ')
103     linked_list.print_data()

```

Implementation of Singly Linked List

```

1  # class of a node
2  class Node:
3      def __init__(self, data):
4          self.data = data
5          self.next = None
6
7
8  # class of the singly linked list
9  class SinglyLinkedList:
10     def __init__(self):
11         self.head = None
12
13     # function to calculate size of the list
14     def size(self):
15         # initialize temporary variable and size to zero
16         current = self.head
17         size = 0
18
19         # count until current does not reach the end of the list i.e. NULL or None
20         while current is not None:
21             size += 1
22             current = current.next
23         return size
24
25     # function to insert at the end of the list
26     def insert_at_end(self, data):
27         # Create the new node
28         node = Node(data)
29
30         # If the list is empty
31         if self.head is None:
32             self.head = node
33         else:
34             current = self.head
35
36             # Otherwise move to the last node of the list
37             while current.next is not None:

```

```
38         current = current.next
39
40         # Point the last node of the list to the new node
41         # So that the new node gets added to the end of the list
42         current.next = node
43
44     # function to insert at the beginning of the list
45     def insert_at_beg(self, data):
46         node = Node(data)
47         # Next pointer of the new node points to the head
48         node.next = self.head
49
50         # Updated the head as new node
51         self.head = node
52
53     # function to delete from the end of the list
54     def delete_from_end(self):
55         current = self.head
56         previous = None
57
58         if current is None:
59             print("Linked List Underflow!!")
60         else:
61             while current.next is not None:
62                 previous = current
63                 current = current.next
64
65             if previous is None:
66                 self.head = None
67             else:
68                 previous.next = None
69
70     # function to delete from the beginning of the list
71     def delete_from_beg(self):
72         current = self.head
73
74         if current is None:
75             print("Linked List Underflow!!")
76         else:
77             self.head = current.next
78
79     # function to print the linked list data
80     def print_data(self):
```

```
81         current = self.head
82
83         while current is not None:
84             print(current.data, '->', end='')
85             current = current.next
86         print('End of list')
87
88
89 # Main program:
90 if __name__ == '__main__':
91     # Create a singly linked list object
92     linked_list = SinglyLinkedList()
93
94     # Insert at the beginning 3, 2, 1
95     linked_list.insert_at_beg(3)
96     linked_list.insert_at_beg(2)
97     linked_list.insert_at_beg(1)
98     print('After insertion at the beginning:')
99     linked_list.print_data()
100
101     # Insert at the end of the list 4, 5
102     linked_list.insert_at_end(4)
103     linked_list.insert_at_end(5)
104     print('After insertion at the end:')
105     linked_list.print_data()
106
107     # Delete 1 from the beginning
108     print('After deletion at the beginning:')
109     linked_list.delete_from_beg()
110     linked_list.print_data()
111
112     # Delete 5 from the end
113     print('After deletion at the end:')
114     linked_list.delete_from_end()
115     linked_list.print_data()
116
117     # print size of the list
118     print("size: ", linked_list.size())
```

Skip M nodes and then delete N nodes alternately


```
1  # Python program to skip M nodes and then delete N nodes alternately in a singly lin\
2  ked list
3
4
5  # Node class
6  class Node:
7
8      # Constructor to initialise data and next
9      def __init__(self, data=None):
10         self.data = data
11         self.next = None
12
13
14  class SinglyLinkedList:
15
16      # Constructor to initialise head
17      def __init__(self):
18         self.head = None
19
20      # Function to skip M delete N nodes
21      def skip_m_delete_n(self, m, n):
22         current = self.head
23
24         # if list is empty return
25         if current is None:
26             return
27
28         # Main loop to traverse the whole list
29         while current is not None:
30             # loop to skip M nodes
31             for i in range(1, m):
32                 if current is None:
33                     return
34                 current = current.next
35
36             if current is None:
37                 return
38
39             # loop to delete N nodes
40             temp = current.next
41             for i in range(1, n+1):
42                 if temp is None:
43                     break
```

```
44         temp = temp.next
45
46         # Point the last node skipped to the node after N nodes deletion
47         current.next = temp
48         # set current for next iteration
49         current = temp
50
51     # Function to Insert data at the beginning of the linked list
52     def insert_at_beg(self, data):
53         node = Node(data)
54         node.next = self.head
55         self.head = node
56
57     # Function to print the linked list
58     def print_data(self):
59         current = self.head
60         while current is not None:
61             print(current.data, '-> ', end='')
62             current = current.next
63         print('None')
64
65 if __name__ == '__main__':
66     linked_list = SinglyLinkedList()
67     linked_list.insert_at_beg(9)
68     linked_list.insert_at_beg(8)
69     linked_list.insert_at_beg(7)
70     linked_list.insert_at_beg(6)
71     linked_list.insert_at_beg(5)
72     linked_list.insert_at_beg(4)
73     linked_list.insert_at_beg(3)
74     linked_list.insert_at_beg(2)
75     linked_list.insert_at_beg(1)
76
77     linked_list.print_data()
78
79     # call the skip_m_delete_n function
80     linked_list.skip_m_delete_n(2, 2)
81
82     # print the modified linked list
83     linked_list.print_data()
```

Chapter 6: Mathematics

Fine the number of trailing zeros in factorial of a number

```
1  # Find the number of trailing zeros present in the factorial of a number n.
2
3  def fact_trailing_zeros(num):
4      c = 5
5      count = 0
6
7      # count number of factors of 5 possible in num!
8      # 5 paired with 2 will give 10 i.e. 1 trailing zero
9      # powers of 5 will give multiple zeros
10     while num // c != 0:
11         count += num // c
12         c *= 5
13
14     return count
15
16 num = 1000
17 print(f"{num}! has {fact_trailing_zeros(num)} trailing zeros")
```

Find the greatest common divisor of 2 numbers

```
1  # Basic eucledian algorithm to find the greatest common divisor of 2 numbers
2
3
4  def gcd(a, b):
5      if a == 0:
6          return b
7      return gcd(b % a, a)
8
9  print(gcd(10, 15))
```

Print all prime factors of a given number

Given a number n, write an efficient function to print all prime factors of n.

For example, if the input number is 12, then output should be “2 2 3”.

```
1  # print all prime factors of a given number
2
3
4  from math import sqrt
5
6
7  def prime_factors(num):
8      # list to store the prime factors
9      prime_factor_lis = []
10
11     # if 2 is a factor of the number
12     while num % 2 == 0:
13         prime_factor_lis.append(2)
14         num /= 2
15
16     for i in range(3, int(sqrt(num)), 2):
17         while num % i == 0:
18             prime_factor_lis.append(i)
19             num /= i
20
21     return prime_factor_lis
22
23 if __name__ == '__main__':
24     print(prime_factors(315))
```

Sieve of Eratosthenes (find prime numbers up to n efficiently)

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so.

```
1  # Given a number n, print all primes smaller than or equal to n. It is also given th\
2  at n is a small number.
3
4  # function to find prime numbers less than or equal to num
5  def find_primes_sieve(num):
6      # list of all numbers upto n
7      intList = [True for i in range(num+1)]
8
9      # first prime
10     p = 2
11
12     while p * p <= num:
13
14         # if intList[p] is True means its a prime number
15         if intList[p]:
16             for i in range(p**2, num+1, p):
17                 intList[i] = False
18
19         p += 1
20
21     lis = []
22     for i in range(2, len(intList)):
23         if intList[i]:
24             lis.append(i)
25
26     return lis
27
28 if __name__ == '__main__':
29     primes = find_primes_sieve(30)
30     print(primes)
```

Chapter 7: Matrix

Given the Coordinates of King and Queen on a chessboard, check if queen threatens the king

Given the Coordinates of King and Queen on a chessboard, check if queen threatens the king.

```
1 def check_threat(king_x, king_y, queen_x, queen_y):
2     # If coordinates are non-integer or outside the bounds of the chessboard
3     if not (validate(king_x) and validate(king_y) and validate(queen_x) and validate\
4 (queen_y)):
5         return False
6
7     # if king is in the vertical column of queen, king_x = queen_x
8     # if king is in the horizontal row of queen, king_y = queen_y
9     # if king is in the diagonal of queen, abs(king_y - queen_y) = abs(king_x - queen_y)
10    n_x) because they will form a square
11    if king_x == queen_x or king_y == queen_y or abs(king_y - queen_y) == abs(king_x -
12 - queen_x):
13        return True
14
15    return False
16
17
18 def validate(coordinate):
19     if type(coordinate) is int and 1 <= coordinate <= 8:
20         return True
21
22     return False
23
24
25 print("Queen threatens King") if check_threat(1, 1, 5, 5) else print("Queen does not\
26 threaten King")
```

Search in a row wise and column wise sorted matrix

Given an $n \times n$ matrix, where every row and column is sorted in increasing order.

Given a number x , how to decide whether this x is in the matrix.

```

1  def search(mat, x):
2      n = len(mat)
3      i = 0
4      j = n-1
5      while i < n and j >= 0:
6          if mat[i][j] == x:
7              return i, j
8          elif mat[i][j] < x:
9              i += 1
10         else:
11             j -= 1
12     return [-1]
13
14
15 matrix = [[10, 20, 30, 40],
16           [15, 25, 35, 45],
17           [27, 29, 37, 48],
18           [32, 33, 39, 50]
19          ]
20
21 print("Enter element you want to search: ")
22 element = int(input())
23 index = search(matrix, element)
24 if len(index) == 1:
25     print("Element not found")
26 else:
27     print("element found at position:(", index[0], ",", index[1], ")")

```

Given a 2D array, print it in spiral form

Input:

```

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

```

Output:

```

1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

```

```
1  # Given a 2D array, print it in spiral form.
2
3  def print_spiral(mat):
4      row, col = len(mat), len(mat[0])
5      k = 0
6      l = 0
7      while k < row and l < col:
8          for i in range(col):
9              print(mat[k][i], end=' ')
10         k += 1
11
12         for i in range(k, row):
13             print(mat[i][col-1], end=' ')
14         col -= 1
15
16         if k < row:
17             for i in reversed(range(l, col-1)):
18                 print(mat[row-1][i], end=' ')
19             row -= 1
20
21         if l < col:
22             for i in reversed(range(k, row-1)):
23                 print(mat[i][l], end=' ')
24             l += 1
25
26
27  a = [[1, 2, 3, 4, 5, 6],
28        [7, 8, 9, 10, 11, 12],
29        [13, 14, 15, 16, 17, 18]
30        ]
31
32  print_spiral(a)
```


Chapter 8: Strings or Arrays

Find the longest substring with k unique characters in a given string

```
1  # Find the longest substring with k unique characters in a given string
2
3
4  def longest_k_unique(string, k):
5      unique = 0
6      sets = set({})
7
8      for i in string:
9          if i not in sets:
10             sets.add(i)
11             unique += 1
12
13     if unique < k:
14         return -1, -1
15
16     count = [0] * 26
17     curr_end = curr_start = max_window_start = 0
18     max_window_len = 1
19
20     count[ord(string[0]) - ord('a')] += 1
21     for i in range(1, len(string)):
22         count[ord(string[i]) - ord('a')] += 1
23         curr_end += 1
24
25         while not isValid(count, k):
26             count[ord(string[curr_start]) - ord('a')] -= 1
27             curr_start += 1
28
29         if curr_end - curr_start + 1 > max_window_len:
30             max_window_len = curr_end - curr_start + 1
31             max_window_start = curr_start
32
33     return max_window_start, max_window_len
```

```

34
35
36 def isValid(count, k):
37     val = 0
38     for i in count:
39         if i > 0:
40             val += 1
41
42     return k >= val
43
44
45 if __name__ == '__main__':
46     string = 'aabaabab'
47     k = 3
48     max_start, max_len = longest_k_unique(string, k)
49
50     if max_len == -1:
51         print("K unique characters sub string does not exist.")
52     else:
53         print('max string with {} unique characters is {}'.format(k) + string[max_start\
54 rt: max_start + max_len] +
55         '" of length', max_len)

```

Find a pattern in a string using KMP search algorithm

```

1  # To find indexes of String "Pattern" in a given String "Text" using KMP Algorithm
2
3
4  # function that returns list of indexes where the patterns matches
5  def KMP_Search(pattern, text):
6      n = len(text)
7      m = len(pattern)
8
9      # pre-compute prefix array of the pattern
10     prefix_arr = get_prefix_arr(pattern, m)
11
12     # stores start point of pattern match in text
13     start_points = []
14
15     i = 0
16     j = 0
17

```

```
18     # while the whole text has not been searched
19     while i != n:
20         # if the character in text matches the pattern character
21         if text[i] == pattern[j]:
22             i += 1
23             j += 1
24         # else find the previous index from where the matching can resume
25         else:
26             j = prefix_arr[j-1]
27
28         # if pattern length has been reached that means a pattern has been found
29         if j == m:
30             start_points.append(i-j)
31             j = prefix_arr[j-1]
32         elif j == 0:
33             i += 1
34         # return the starting position of pattern in text
35     return start_points
36
37
38 # pre-computes the prefix array for KMP search
39 def get_prefix_arr(pattern, m):
40     prefix_arr = [0] * m
41     j = 0
42     i = 1
43
44     while i != m:
45         if pattern[i] == pattern[j]:
46             j += 1
47             prefix_arr[i] = j
48             i += 1
49         elif j != 0:
50             j = prefix_arr[j-1]
51         else:
52             prefix_arr[i] = 0
53             i += 1
54
55     return prefix_arr
56
57 txt = "ABABDABACDABABCABABCABAB"
58 pat = "ABABCABAB"
59
60 start_indexes = KMP_Search(pat, txt)
```

```
61
62 for i in start_indexes:
63     print(i)
```

Find the Kth smallest element in the array

Approach used: QuickSelect

Time Complexity: O(n)

```
1  # Find the Kth smallest element in a given array.
2  # taking smallest in arr as 1st smallest
3
4  def partition(arr, low, high):
5      i = (low - 1)
6      pivot = arr[high] # pivot
7
8      for j in range(low, high):
9
10         # If current element is smaller than or
11         # equal to pivot
12         if arr[j] <= pivot:
13             # increment index of smaller element
14             i += 1
15             arr[i], arr[j] = arr[j], arr[i]
16
17     arr[i + 1], arr[high] = arr[high], arr[i + 1]
18     return i + 1
19
20
21 def quick_select(arr, low, high, k):
22     # arr follows zero indexing hence kth smallest will be at index (k - 1)
23     k -= 1
24     while low < high:
25         p_index = partition(arr, low, high)
26
27         # found the kth smallest value
28         if p_index == k:
29             return arr[p_index]
30         # pivot index is less than k hence kth smallest is in the right half
31         elif p_index < k:
32             low = p_index + 1
33         # pivot index is greater than k hence kth smallest is in the left half
```

```

34         else:
35             high = p_index - 1
36             # if k < 0 or k > len(arr) then simply return the smallest or largest value in a\
37 rr
38         return arr[low]
39
40
41 arr = [10, 7, 8, 9, 1, 5]
42 n = len(arr) - 1
43
44 # find 4th smallest element in the array
45 print(quick_select(arr, 0, n, 4))

```

Find a pair in an array with sum x

```

1  # Find a pair of elements in the array with sum = x
2
3  """
4  Method 1: If unsorted array
5  Time Complexity: O(n)
6  Space Complexity: O(n)
7  """
8
9
10 def find_pair_unsorted(arr, x):
11     elem_set = set({})
12
13     # To store the indexes of both the elements
14     pair = [-1, -1]
15
16     for value in arr:
17         # if x - value has already been discovered in the array
18         # Pair found, return the values
19         if (x-value) in elem_set:
20             return x-value, value
21
22         # else add the current value in the elem_set
23         else:
24             elem_set.add(value)
25
26     return "Not found"
27

```

```

28 arr = [1, 4, 45, 6, 10, 8]
29 print('Unsorted array:', arr)
30 print('Pair with sum 16 in unsorted array:', find_pair_unsorted(arr, 16))
31
32
33 """
34 Method 2: If array is sorted
35 Time Complexity: O(n)
36 Space Complexity: O(1)
37 """
38
39
40 def find_pair_sorted(arr, x):
41     # initialize variables to the start and end of the array
42     l = 0
43     r = len(arr) - 1
44
45     while l < r:
46         pair_sum = arr[l] + arr[r]
47
48         # if pair is found
49         if pair_sum == x:
50             return arr[l], arr[r]
51         # if the pair sum is less than x go to the next bigger value from left
52         elif pair_sum < x:
53             l += 1
54         # if the pair sum is more than x go to the next lesser value from right
55         else:
56             r -= 1
57
58     # If pair not found
59     return "Not found"
60
61
62 arr = [2, 6, 10, 15, 18, 20, 23, 25]
63 print('Sorted array:', arr)
64 print('Pair with sum 28 in sorted array:', find_pair_sorted(arr, 28))

```

Print all valid (properly opened and closed) combinations of n pairs of parentheses

Print all valid (properly opened and closed) combinations of n pairs of parentheses.

```

1  def addParentheses(str_arr, leftParenCount, rightParenCount, combinations, index):
2      if leftParenCount < 0 or rightParenCount < 0:
3          return
4      if leftParenCount == 0 and rightParenCount == 0:
5          combinations.append(''.join(str_arr))
6      else:
7          if leftParenCount > 0:
8              str_arr[index] = '('
9              addParentheses(str_arr, leftParenCount - 1, rightParenCount, combination\
10 s, index + 1)
11
12          if rightParenCount > leftParenCount:
13              str_arr[index] = ')'
14              addParentheses(str_arr, leftParenCount, rightParenCount - 1, combination\
15 s, index + 1)
16
17
18 def generateParentheses(count):
19     str_arr = [''] * (count * 2)
20     combinations = []
21     addParentheses(str_arr, count, count, combinations, 0)
22     return combinations
23
24
25 parenthesis_pairs = 3
26 combinations = generateParentheses(parenthesis_pairs)
27 print(*combinations, sep=', ')

```

Reverse the order of the words in the array

['p', 'e', 'r', 'f', 'e', 'c', 't', ' ', 'm', 'a', 'k', 'e', 's', ' ', 'p', 'r', 'a', 'c', 't', 'i', 'c', 'e']

would turn into:

['p', 'r', 'a', 'c', 't', 'i', 'c', 'e', ' ', 'm', 'a', 'k', 'e', 's', ' ', 'p', 'e', 'r', 'f', 'e', 'c', 't']

```

1  def reverse_sentence(arr):
2      # reverse all characters:
3      n = len(arr)
4      mirrorReverse(arr, 0, n-1)
5
6      # reverse each word:
7      word_start = None
8      for i in range(0, n):
9          if arr[i] == " ":
10             if word_start is not None:
11                 mirrorReverse(arr, word_start, i-1)
12                 word_start = None
13             elif i == n-1:
14                 if word_start is not None:
15                     mirrorReverse(arr, word_start, i)
16             else:
17                 if word_start is None:
18                     word_start = i
19
20
21 def mirrorReverse(arr, start, end):
22     while start < end:
23         tmp = arr[start]
24         arr[start] = arr[end]
25         arr[end] = tmp
26         start += 1
27         end -= 1
28
29
30 if __name__ == '__main__':
31     arr = ['p', 'e', 'r', 'f', 'e', 'c', 't', ' ', 'm', 'a', 'k', 'e', 's', ' ', 'p'\
32 , 'r', 'a', 'c', 't', 'i', 'c', 'e']
33
34     print('Before reverse:', ''.join(arr))
35     reverse_sentence(arr)
36     print('After reverse: ', ''.join(arr))

```

Find index of given number in a sorted array shifted by an unknown offset

If the sorted array `arr` is shifted left by an unknown offset and you don't have a pre-shifted copy of it, how would you modify your method to find a number in the shifted array?


```
1 def binarySearch(arr, num, begin, end):
2     while begin <= end:
3         mid = round((begin+end)/2)
4         if arr[mid] < num:
5             begin = mid + 1
6         elif arr[mid] == num:
7             return mid
8         else:
9             end = mid - 1
10    return -1
11
12
13 def shiftedArrSearch(shiftArr, num):
14     originalFirst = getOrigFirst(shiftArr)
15     n = len(shiftArr)
16     if shiftArr[originalFirst] <= num <= shiftArr[n-1]:
17         return binarySearch(shiftArr, num, originalFirst, n - 1)
18     else:
19         return binarySearch(shiftArr, num, 0, originalFirst - 1)
20
21
22 def getOrigFirst(arr):
23     begin = 0
24     end = len(arr)-1
25     while begin <= end:
26         mid = int((begin+end)/2)
27         if mid == 0 or arr[mid] < arr[mid-1]:
28             return mid
29         if arr[mid] > arr[0]:
30             begin = mid + 1
31         else:
32             end = mid - 1
33     return 0
34
35 if __name__ == '__main__':
36     shiftArr = [9, 12, 17, 2, 4, 5]
37     print(shiftedArrSearch(shiftArr, 4))
```

Print all permutations of a given string

```
1  # To print all permutations of a given string
2
3  count = 0
4
5
6  def permutations(mat, l, r):
7      if l == r:
8          print(''.join(mat))
9          global count
10         count += 1
11     else:
12         for i in range(l, r+1):
13             mat[l], mat[i] = mat[i], mat[l]
14             permutations(mat, l+1, r)
15             mat[l], mat[i] = mat[i], mat[l]
16
17
18  string = "ABC"
19  permutations(list(string), 0, len(string)-1)
20  print('total permutations:', count)
```

Linear Search in an array

```
1  # Function for linear search
2  # inputs: array of elements 'arr', key to be searched 'x'
3  # returns: index of first occurrence of x in arr
4  def linear_search(arr, x):
5      # traverse the array
6      for i in range(0, len(arr)):
7
8          # if element at current index is same as x
9          # return the index value
10         if arr[i] == x:
11             return i
12
13     # if the element is not found in the array return -1
14     return -1
15
16  arr = [3, 2, 1, 5, 6, 4]
17  print(linear_search(arr, 1))
```

Binary Search in an array

```
1  # Function for binary search
2  # inputs: sorted array 'arr', key to be searched 'x'
3  # returns: index of the occurrence of x found in arr
4
5
6  def binary_search(arr, x):
7      l = 0
8      r = len(arr)
9
10     # while the left index marker < right index marker
11     while l < r:
12         # find the index of the middle element
13         mid = int(l + ((r - l) / 2))
14
15         # if middle element is x, return mid
16         if arr[mid] == x:
17             return mid
18
19         # if middle element is < x, update l to search to the right of mid
20         elif arr[mid] < x:
21             l = mid + 1
22
23         # if middle element is > x, update r to search to the left of mid
24         else:
25             r = mid - 1
26
27     return -1
28
29
30 arr = [1, 4, 5, 7, 8, 10, 13, 15]
31 print(binary_search(arr, 5))
```

Interpolation Search in an array

```

1  def interpolation_search(arr, key):
2      low = 0
3      high = len(arr) - 1
4
5      while arr[high] != arr[low] and key >= arr[low] and key <= arr[high]:
6          mid = int(low + ((key - arr[low]) * (high - low) / (arr[high] - arr[low])))
7
8          if arr[mid] == key:
9              return mid
10         elif arr[mid] < key:
11             low = mid + 1
12         else:
13             high = mid - 1
14
15     return -1
16
17
18  # input arr
19  arr = [2, 4, 6, 8, 10, 12, 14, 16]
20
21  # interpolation_search call to search 3 in arr
22  print('6 is at index: ', interpolation_search(arr, 6))
23
24  # Output: 6 is at index:  2

```

Bubble sort Algorithm

Output:

Before sorting: [64, 34, 25, 12, 22, 11, 90]

After sorting: [11, 12, 22, 25, 34, 64, 90]

```

1  # bubble sort function
2  def bubble_sort(arr):
3      n = len(arr)
4
5      # Repeat loop N times
6      # equivalent to: for(i = 0; i < n-1; i++)
7      for i in range(0, n-1):
8          # Repeat internal loop for (N-i)th largest element
9          for j in range(0, n-i-1):
10
11              # if jth value is greater than (j+1) value

```

```
12         if arr[j] > arr[j+1]:
13             # swap the values at j and j+1 index
14             # Pythonic way to swap 2 variable values -> x, y = y, x
15             arr[j], arr[j+1] = arr[j+1], arr[j]
16
17
18 arr = [64, 34, 25, 12, 22, 11, 90]
19 print('Before sorting:', arr)
20
21 # call bubble sort function on the array
22 bubble_sort(arr)
23
24 print('After sorting:', arr)
```

Counting sort Algorithm (non-comparison based sorting)

```
1  # counting sort without stable sorting
2
3
4  def counting_sort(arr):
5      n = len(arr)
6
7      # get the maximum value of the array
8      max_val = max(arr)
9
10     count = [0] * (max_val + 1)
11
12     # fill the count array
13     # for each element x in the array
14     for x in arr:
15         count[x] += 1
16
17     k = 0
18     for i in range(0, len(count)):
19         for j in range(0, count[i]):
20             arr[k] = i
21             k += 1
22
23
24 arr = [3, 2, 1, 3, 2, 5, 5, 3]
```

```

25 print('Before counting sort:', arr)
26
27 # call counting sort function
28 counting_sort(arr)
29
30 print('After counting sort:', arr)

```

Insertion sort Algorithm

Output:

Before sort arr: [12, 11, 13, 5, 6]

Sorted arr: [5, 6, 11, 12, 13]

```

1  def insertion_sort(arr):
2      n = len(arr)
3
4      for i in range(1, n):
5          x = arr[i]
6          j = i - 1
7          while j >= 0 and arr[j] > x:
8              # copy value of previous index to index + 1
9              arr[j + 1] = arr[j]
10             # j = j - 1
11             j -= 1
12             # copy the value which was at ith index to its correct sorted position
13             arr[j + 1] = x
14
15
16 arr = [12, 11, 13, 5, 6]
17 print('Before sort arr: ', arr)
18 insertion_sort(arr)
19 print('Sorted arr: ', arr)

```

Sort an array where each element is at most k places away from its sorted position

Given an array arr of length n where each element is at most k places away from its sorted position, Code an efficient algorithm to sort arr.

```

1  import heapq
2
3
4  def kHeapSort(arr, k):
5      h = []
6      n = len(arr)
7      for i in range(0, k+1):
8          heapq.heappush(h, arr[i])
9      for i in range(k+1, n):
10         arr[i-(k+1)] = heapq.heappop(h)
11         heapq.heappush(h, arr[i])
12     for i in range(0, k+1):
13         arr[n-k-1 + i] = heapq.heappop(h)
14     return arr
15
16 if __name__ == '__main__':
17     arr = [2, 1, 4, 3, 6, 5]
18
19     print(kHeapSort(arr, 2))

```

Merge Sort Algorithm

```

1  # Program to perform merge sort on an array
2
3
4  def merge(arr, low, mid, high):
5      n1 = mid - low + 1
6      n2 = high - mid
7
8      # create temporary arrays
9      """
10     arr = [0] * n is equivalent to:
11     arr = [0, 0, 0, ..., 0]
12     array of n zeros
13     """
14     arr1 = [0] * n1
15     arr2 = [0] * n2
16
17     # copy data of arr into arr1 and arr2
18     for i in range(0, n1):
19         arr1[i] = arr[low + i]
20

```

```
21     for i in range(0, n2):
22         arr2[i] = arr[mid + 1 + i]
23
24     # initialize i, j to 0
25     i = j = 0
26
27     # initialize k to lower index
28     k = low
29
30     # merge the 2 arrays
31     while i < n1 and j < n2:
32         if arr1[i] < arr2[j]:
33             arr[k] = arr1[i]
34             i += 1
35         else:
36             arr[k] = arr2[j]
37             j += 1
38         k += 1
39
40     # if elements left in arr1 copy them to arr
41     while i < n1:
42         arr[k] = arr1[i]
43         i += 1
44         k += 1
45
46     # if elements left in arr2 copy them to arr
47     while j < n2:
48         arr[k] = arr2[j]
49         j += 1
50         k += 1
51
52
53 def merge_sort(arr, low, high):
54     if low < high:
55         # mid = int((low + high) / 2)
56         mid = int(low + ((high - low) / 2))
57
58         # call merge_sort on 2 halves
59         merge_sort(arr, low, mid)
60         merge_sort(arr, mid+1, high)
61
62         # merge the two sorted halves
63         merge(arr, low, mid, high)
```



```
64
65
66 arr = [5, 21, 7, 3, 4, 8, 9, 10, 100, 15]
67
68 print('Before merge sort:', arr)
69
70 # Call merge sort on arr
71 merge_sort(arr, 0, len(arr)-1)
72
73 print('After merge sort:', arr)
```

Quick Sort Algorithm using last element as pivot

```
1  # Python program for implementation of Quicksort Sort by taking last element as pivot
2
3
4  def partition(arr, low, high):
5      i = (low - 1)
6      pivot = arr[high] # pivot
7
8      for j in range(low, high):
9
10         # If current element is smaller than or
11         # equal to pivot
12         if arr[j] <= pivot:
13             # increment index of smaller element
14             i += 1
15             arr[i], arr[j] = arr[j], arr[i]
16
17     arr[i + 1], arr[high] = arr[high], arr[i + 1]
18     return i + 1
19
20
21 # Function to do Quick sort
22 def quickSort(arr, low, high):
23     if low < high:
24         # pivot is set to its right position after partition call
25         pi = partition(arr, low, high)
26
27         # Separately sort elements before
28         # partition and after partition
29         quickSort(arr, low, pi - 1)
```

```
30         quickSort(arr, pi + 1, high)
31
32
33 # Driver code to test above
34 arr = [10, 7, 8, 9, 1, 5]
35 n = len(arr)
36 print("Before sorting array is:")
37 for i in range(n):
38     print(arr[i], end=' -> ')
39 print('end')
40
41 quickSort(arr, 0, n - 1)
42
43 print("Sorted array is:")
44 for i in range(n):
45     print(arr[i], end=' -> ')
46 print('end')
```

Selection sort Algorithm

```
1 # selection sort function
2 def selection_sort(arr):
3     n = len(arr)
4     for i in range(0, n):
5         for j in range(i+1, n):
6             # if the value at i is > value at j -> swap
7             if arr[i] > arr[j]:
8                 arr[i], arr[j] = arr[j], arr[i]
9
10
11 # input arr
12 arr = [3, 2, 4, 1, 5]
13 print('Before selection sort:', arr)
14
15 # call selection sort function
16 selection_sort(arr)
17 print('After selection sort:', arr)
```

Chapter 9: Tree

Binary Search Tree implementation

```
1  #!/usr/bin/python3
2
3
4  class Node:
5      def __init__(self, data=None, right_child=None, left_child=None, parent=None):
6          self.data = data
7          self.right_child = right_child
8          self.left_child = left_child
9          self.parent = parent
10
11     def has_left_child(self):
12         return self.left_child
13
14     def has_right_child(self):
15         return self.right_child
16
17     def has_both_children(self):
18         return self.right_child and self.left_child
19
20     def has_any_children(self):
21         return self.right_child or self.left_child
22
23     def is_root(self):
24         return not self.parent
25
26     def is_leaf(self):
27         return not (self.right_child or self.left_child)
28
29     def is_left_child(self):
30         return self.parent and self.parent.left_child == self
31
32     def is_right_child(self):
33         return self.parent and self.parent.right_child == self
34
35     def replace_data(self, data):
```

```
36         self.data = data
37
38     def replace_left_child(self, left_child):
39         self.left_child = left_child
40
41     def replace_right_child(self, right_child):
42         self.right_child = right_child
43
44
45 class BinarySearchTree:
46     def __init__(self):
47         self.root = None
48
49     def insert_node(self, data):
50         if self.root is None:
51             self.root = Node(data)
52         else:
53             self._insert(data, self.root)
54
55     def _insert(self, data, current_node):
56         if data < current_node.data:
57             if current_node.has_left_child():
58                 self._insert(data, current_node.left_child)
59             else:
60                 current_node.left_child = Node(data, parent=current_node)
61         else:
62             if current_node.has_right_child():
63                 self._insert(data, current_node.right_child)
64             else:
65                 current_node.right_child = Node(data, parent=current_node)
66
67     @staticmethod
68     def find_inorder_ancestor(current_node):
69         if current_node.has_right_child():
70             current_node = current_node.right_child
71         while current_node.has_left_child():
72             current_node = current_node.left_child
73         return current_node
74
75     ancestor = current_node.parent
76     child = current_node
77     while ancestor is not None and child is ancestor.right_child:
78         child = ancestor
```

```
79         ancestor = child.parent
80     return ancestor
81
82     def delete_node(self, data):
83         if self.root is None:
84             print("No node to delete in the tree")
85         else:
86             if self._delete(data, self.root) is not None:
87                 print("Deleted", data)
88             else:
89                 print(data, "not found")
90
91     def _delete(self, data, current_node):
92         while current_node is not None and data != current_node.data:
93             if data <= current_node.data:
94                 current_node = current_node.left_child
95             else:
96                 current_node = current_node.right_child
97
98         if current_node is not None:
99             if current_node.is_leaf():
100                 if current_node.is_left_child():
101                     current_node.parent.left_child = None
102                 else:
103                     current_node.parent.right_child = None
104             else:
105                 successor = self.find_inorder_ancestor(current_node)
106                 if successor is None:
107                     current_node.data = current_node.left_child.data
108                     current_node.left_child = None
109                 else:
110                     temp = current_node.data
111                     current_node.data = successor.data
112                     successor.data = temp
113                     self._delete(temp, successor)
114             return True
115         else:
116             return None
117
118     def inorder(self):
119         current_node = self.root
120         self._inorder(current_node)
121         print('End')
```

```
122
123     def _inorder(self, current_node):
124         if current_node is None:
125             return
126         self._inorder(current_node.left_child)
127         print(current_node.data, " -> ", end='')
128         self._inorder(current_node.right_child)
129
130 if __name__ == '__main__':
131     tree = BinarySearchTree()
132     tree.insert_node(6)
133     tree.insert_node(9)
134     tree.insert_node(6)
135     tree.insert_node(5)
136     tree.insert_node(8)
137     tree.insert_node(7)
138     tree.insert_node(3)
139     tree.insert_node(2)
140     tree.insert_node(4)
141     tree.inorder()
142     tree.delete_node(6)
143     tree.delete_node(1)
144     tree.delete_node(3)
145     tree.inorder()
```

Check if a given array can represent Preorder Traversal of Binary Search Tree

```
1  # Check if a given array can represent Preorder Traversal of Binary Search Tree
2
3
4  def check_pre_order(arr):
5      minimum = -float('inf')
6
7      stack = []
8
9      for val in arr:
10         if val < minimum:
11             return False
12
13         while len(stack) != 0 and stack[-1] < val:
```

```

14         minimum = stack.pop()
15
16         stack.append(val)
17
18     return True
19
20
21 if __name__ == '__main__':
22     pre_order = [40, 30, 35, 80, 100]
23     print("Valid Preorder" if check_pre_order(pre_order) else "Invalid Preorder")
24     pre_order = [40, 30, 35, 20, 80, 100]
25     print("Valid Preorder" if check_pre_order(pre_order) else "Invalid Preorder")

```

Find the in-order ancestor of a given node in BST

```

1  # find the in-order ancestor of a given node in BST
2
3
4  # A Binary Search Tree node
5  class Node:
6      # Constructor to initialise node
7      def __init__(self, data, parent=None):
8          self.data = data
9          self.left = None
10         self.right = None
11         self.parent = parent
12
13
14  def findInOrderAncestor(n):
15      if n.right is not None:
16          return findMinKeyWithinTree(n.right)
17
18      ancestor = n.parent
19      child = n
20      while ancestor is not None and child == ancestor.right:
21          child = ancestor
22          ancestor = child.parent
23      return ancestor
24
25
26  def findMinKeyWithinTree(root):
27      while root.left is not None:

```

```

28         root = root.left
29     return root
30
31
32 if __name__ == '__main__':
33     root = Node(4)
34     root.left = Node(2, root)
35     root.right = Node(6, root)
36     root.left.left = Node(1, root.left)
37     root.left.right = Node(3, root.left)
38     successor = findInOrderAncestor(root.left.right)
39
40     if successor is not None:
41         print(successor.data)
42     else:
43         print("No in order successor")

```

Find the Lowest Common Ancestor

```

1  # Find the Lowest Common Ancestor (LCA) in a Binary Search Tree
2
3
4  # A Binary Search Tree node
5  class Node:
6      # Constructor to initialise node
7      def __init__(self, data):
8          self.data = data
9          self.left = None
10         self.right = None
11
12
13 class BST:
14     def __init__(self):
15         self.root = None
16
17     def insert_node(self, data):
18         if self.root is None:
19             self.root = Node(data)
20         else:
21             self._insert(data, self.root)
22
23     def _insert(self, data, current_node):

```



```
24         if data <= current_node.data:
25             if current_node.left is not None:
26                 self._insert(data, current_node.left)
27             else:
28                 current_node.left = Node(data)
29         else:
30             if current_node.right is not None:
31                 self._insert(data, current_node.right)
32             else:
33                 current_node.right = Node(data)
34
35     def inorder(self):
36         current_node = self.root
37         self._inorder(current_node)
38         print('End')
39
40     def _inorder(self, current_node):
41         if current_node is None:
42             return
43         self._inorder(current_node.left)
44         print(current_node.data, " -> ", end='')
45         self._inorder(current_node.right)
46
47
48     # assuming both nodes are present in the tree
49     def lca_bst(root, value1, value2):
50         while root is not None:
51             if value2 > root.data < value1:
52                 root = root.right
53             elif value2 < root.data > value1:
54                 root = root.left
55             else:
56                 return root.data
57
58
59 if __name__ == '__main__':
60     tree = BST()
61     tree.insert_node(6)
62     tree.insert_node(8)
63     tree.insert_node(9)
64     tree.insert_node(6)
65     tree.insert_node(5)
66     tree.insert_node(7)
```

```

67     tree.insert_node(3)
68     tree.insert_node(2)
69     tree.insert_node(4)
70     print(lca_bst(tree.root, 4, 2))
71
72     """
73     given tree:
74           6
75        6   8
76     5   7   9
77    3
78   2   4
79     """

```

Given a sorted array, create a BST with minimal height

```

1  # Given a sorted array, create a binary search tree with minimal height
2
3
4  # A Binary Tree node
5  class Node:
6      # Constructor to initialise node
7      def __init__(self, data):
8          self.data = data
9          self.left = None
10         self.right = None
11
12
13  def make_minimal_bst(arr, start, end):
14      if end < start:
15          return None
16      mid = int((start + end) / 2)
17      root = Node(arr[mid])
18
19      root.left = make_minimal_bst(arr, start, mid-1)
20      root.right = make_minimal_bst(arr, mid+1, end)
21
22      return root
23
24
25  def in_order(root):
26      if root is None:

```

```

27         return
28     in_order(root.left)
29     print(root.data, end=' -> ')
30     in_order(root.right)
31
32 if __name__ == '__main__':
33     arr = [1, 2, 3, 4, 5, 6, 7, 8]
34     root = make_minimal_bst(arr, 0, len(arr)-1)
35     in_order(root)
36     print('end')

```

Print Nodes in Bottom View of Binary Tree

```

1  # Print Nodes in Bottom View of Binary Tree
2  from collections import deque
3
4
5  class Node:
6      def __init__(self, data):
7          self.data = data
8          self.left = None
9          self.right = None
10
11
12 def bottom_view(root):
13     if root is None:
14         return
15
16     # make an empty queue for BFS
17     q = deque()
18
19     # dict to store bottom view keys
20     bottomview = {}
21
22     # append root in the queue with horizontal distance as 0
23     q.append((root, 0))
24
25     while q:
26         # get the element and horizontal distance
27         elem, hd = q.popleft()
28
29         # update the last seen hd element

```

```

30         bottomview[hd] = elem.data
31
32         # add left and right child in the queue with hd - 1 and hd + 1
33         if elem.left is not None:
34             q.append((elem.left, hd - 1))
35         if elem.right is not None:
36             q.append((elem.right, hd + 1))
37
38         # return the bottomview
39         return bottomview
40
41
42 if __name__ == '__main__':
43     root = Node(20)
44     root.left = Node(8)
45     root.right = Node(22)
46     root.left.left = Node(5)
47     root.left.right = Node(3)
48     root.right.left = Node(4)
49     root.right.right = Node(25)
50     root.left.right.left = Node(10)
51     root.left.right.right = Node(14)
52
53     bottomview = bottom_view(root)
54
55     for i in sorted(bottomview):
56         print(bottomview[i], end=' ')

```

Check if a binary tree is height balanced

```

1  # Check if a binary tree is height balanced
2  # abs(height[leftTree] - height[rightTree]) <= 1
3
4
5  # A Binary Tree node
6  class Node:
7      # Constructor to initialise node
8      def __init__(self, data):
9          self.data = data
10         self.left = None
11         self.right = None
12

```

```
13
14 def check_height(root):
15     if root is None:
16         return -1
17     left_height = check_height(root.left)
18     if left_height is float('inf'):
19         return float('inf')
20
21     right_height = check_height(root.right)
22     if right_height is float('inf'):
23         return float('inf')
24
25     height = abs(left_height - right_height)
26     if height > 1:
27         return float('inf')
28     else:
29         return max(left_height, right_height) + 1
30
31
32 def isBalanced(root):
33     return check_height(root) != float('inf')
34
35 if __name__ == '__main__':
36     root = Node(1)
37     root.left = Node(2)
38     root.right = Node(3)
39     root.left.left = Node(4)
40     root.left.right = Node(5)
41
42     if isBalanced(root):
43         print("Yes! the tree is balanced")
44     else:
45         print("No! the tree is not balanced")
```

Check whether a binary tree is a full binary tree or not

```
1  # Check whether a binary tree is a full binary tree or not
2
3
4  class Node:
5      def __init__(self, data):
6          self.data = data
7          self.left = None
8          self.right = None
9
10
11 def check_full_bt(root):
12     # If tree is empty
13     if root is None:
14         return True
15
16     # if both child are None
17     if root.left is None and root.right is None:
18         return True
19
20     # if the node has both children and both sub tree are full binary trees
21     if root.left is not None and root.right is not None:
22         return check_full_bt(root.left) and check_full_bt(root.right)
23
24     return False
25
26 if __name__ == '__main__':
27     root = Node(10)
28     root.left = Node(20)
29     root.right = Node(30)
30
31     root.left.right = Node(40)
32     root.left.left = Node(50)
33     root.right.left = Node(60)
34     root.right.right = Node(70)
35
36     root.left.left.left = Node(80)
37     root.left.left.right = Node(90)
38     root.left.right.left = Node(80)
39     root.left.right.right = Node(90)
40     root.right.left.left = Node(80)
41     root.right.left.right = Node(90)
42     root.right.right.left = Node(80)
43     root.right.right.right = Node(90)
```

```
44
45     if check_full_bt(root):
46         print('Yes, given binary tree is Full BT')
47     else:
48         print('No, given binary tree is not Full BT')
```

Given two binary trees, check if the first tree is subtree of the second one

```
1  # Given two binary trees, check if the first tree is subtree of the second one.
2
3
4  # A Binary Tree node
5  class Node:
6      # Constructor to initialise node
7      def __init__(self, data):
8          self.data = data
9          self.left = None
10         self.right = None
11
12
13  def store_in_order(root, arr):
14      if root is None:
15          arr.append('$')
16          return
17      store_in_order(root.left, arr)
18      arr.append(root.data)
19      store_in_order(root.right, arr)
20
21
22  def store_pre_order(root, arr):
23      if root is None:
24          arr.append('$')
25          return
26      store_pre_order(root.left, arr)
27      store_pre_order(root.right, arr)
28      arr.append(root.data)
29
30
31  def isSubTree(t1, t2):
32      order_t1 = []
```

```

33     order_t2 = []
34
35     store_in_order(t1, order_t1)
36     store_in_order(t2, order_t2)
37     if ''.join(order_t1).find(''.join(order_t2)) == -1:
38         return False
39
40     order_t1 = []
41     order_t2 = []
42
43     store_pre_order(t1, order_t1)
44     store_pre_order(t2, order_t2)
45     if ''.join(order_t1).find(''.join(order_t2)) == -1:
46         return False
47
48     return True
49
50
51 T = Node('a')
52 T.left = Node('b')
53 T.right = Node('d')
54 T.left.left = Node('c')
55 T.left.right = Node('e')
56
57 S = Node('b')
58 S.left = Node('c')
59 S.right = Node('e')
60
61 if isSubTree(T, S):
62     print('Yes, S is a subtree of T')
63 else:
64     print('No, S is not a subtree of T')

```

Find the Lowest Common Ancestor in a Binary Tree

Program to find LCA of n1 and n2 using one traversal of Binary tree


```
1  # A binary tree node
2  class Node:
3      # Constructor to create a new node
4      def __init__(self, key):
5          self.key = key
6          self.left = None
7          self.right = None
8
9
10 # This function returns reference to LCA of two given values n1 and n2
11 # v1 is set as true by this function if n1 is found
12 # v2 is set as true by this function if n2 is found
13 def findLCAUtil(root, n1, n2, v):
14     # Base Case
15     if root is None:
16         return None
17
18     # IF either n1 or n2 matches ith root's key, report
19     # the presence by setting v1 or v2 as true and return
20     # root
21     if root.key == n1:
22         v[0] = True
23         return root
24
25     if root.key == n2:
26         v[1] = True
27         return root
28
29     # Look for keys in left and right subtree
30     left_lca = findLCAUtil(root.left, n1, n2, v)
31     right_lca = findLCAUtil(root.right, n1, n2, v)
32
33     # if one key is present in left subtree and other is present in other,
34     # So this node is the LCA
35     if left_lca and right_lca:
36         return root
37
38     # Otherwise check if left subtree or right subtree is LCA
39     return left_lca if left_lca is not None else right_lca
40
41
42 def find(root, k):
43     # Base Case
```

```
44     if root is None:
45         return False
46
47     # If key is present at root, or if left subtree or right
48     # subtree , return true
49     if (root.key == k or find(root.left, k) or
50         find(root.right, k)):
51         return True
52
53     # Else return false
54     return False
55
56
57 # This function returns LCA of n1 and n2 only if both
58 # n1 and n2 are present in tree, otherwise returns None
59 def findLCA(root, n1, n2):
60     # Initialize n1 and n2 as not visited
61     v = [False, False]
62
63     # Find lca of n1 and n2
64     lca = findLCAUtil(root, n1, n2, v)
65
66     # Returns LCA only if both n1 and n2 are present in tree
67     if v[0] and v[1] or v[0] and find(lca, n2) or v[1] and find(lca, n1):
68         return lca
69
70     # Else return None
71     return None
72
73
74 # Driver program to test above function
75 root = Node(1)
76 root.left = Node(2)
77 root.right = Node(3)
78 root.left.left = Node(4)
79 root.left.right = Node(5)
80 root.right.left = Node(6)
81 root.right.right = Node(7)
82
83 lca = findLCA(root, 4, 5)
84
85 if lca is not None:
86     print("LCA(4, 5) = ", lca.key)
```

```
87 else:
88     print("Keys are not present")
89
90 lca = findLCA(root, 4, 10)
91 if lca is not None:
92     print("LCA(4,10) = ", lca.key)
93 else:
94     print("Keys are not present")
```

Create a list of all nodes at each depth

```
1  # Create a list of all nodes at each depth
2
3  # A Binary Tree node
4  class Node:
5      # Constructor to initialise node
6      def __init__(self, data):
7          self.data = data
8          self.left = None
9          self.right = None
10
11
12 def list_of_depths(root):
13     if root is None:
14         return []
15     depths = []
16     q = []
17     q.append(root)
18
19     while q:
20         parents = q
21         depths.append(q)
22         q = []
23         for parent in parents:
24             if parent.left is not None:
25                 q.append(parent.left)
26             if parent.right is not None:
27                 q.append(parent.right)
28
29     return depths
30
31 if __name__ == '__main__':
```

```

32     root = Node(1)
33     root.left = Node(2)
34     root.right = Node(3)
35     root.left.left = Node(4)
36     root.left.right = Node(5)
37
38     for depth, list_nodes in enumerate(list_of_depths(root)):
39         print('Depth', depth, end=': ')
40         for n in list_nodes:
41             print(n.data, end=' -> ')
42         print('end')

```

Find the maximum path sum i.e. max sum of a path in a binary tree

```

1  # find the maximum path sum. The path may start and end at any node in the tree.
2
3  class Node:
4      def __init__(self, data):
5          self.data = data
6          self.left = None
7          self.right = None
8
9
10 def max_sum_path(root):
11     max_sum_path_util.res = -float('inf')
12
13     max_sum_path_util(root)
14     return max_sum_path_util.res
15
16
17 def max_sum_path_util(root):
18     if root is None:
19         return 0
20     # find max sum in left and right sub tree
21     left_sum = max_sum_path_util(root.left)
22     right_sum = max_sum_path_util(root.right)
23
24     # if current node is one of the nodes in the path above for max
25     # it can either be alone, or with left sub tree or right sub tree
26     max_single = max(max(left_sum, right_sum) + root.data, root.data)

```

```

27
28     # if the current root itself is considered as top node of the max path
29     max_parent = max(left_sum + right_sum + root.data, max_single)
30
31     # store the maximum result
32     max_sum_path_util.res = max(max_sum_path_util.res, max_parent)
33
34     # return the max_single for upper nodes calculation
35     return max_single
36
37
38 if __name__ == '__main__':
39     root = Node(10)
40     root.left = Node(2)
41     root.right = Node(10)
42     root.left.left = Node(20)
43     root.left.right = Node(1)
44     root.right.right = Node(-25)
45     root.right.right.left = Node(3)
46     root.right.right.right = Node(4)
47
48     print('max path sum is:', max_sum_path(root))

```

Find minimum depth of a binary tree

```

1  # Find minimum depth of a binary tree
2  from collections import deque
3
4
5  # A Binary Tree node
6  class Node:
7      # Constructor to initialise node
8      def __init__(self, data):
9          self.data = data
10         self.left = None
11         self.right = None
12
13
14 def get_min_depth(root):
15     if root is None:
16         return 0
17

```

```

18     queue = deque()
19     queue.append((root, 1))
20
21     while queue:
22         node, height = queue.popleft()
23         if node.left is None and node.right is None:
24             return height
25
26         else:
27             if node.left is not None:
28                 queue.append((node.left, height + 1))
29             if node.right is not None:
30                 queue.append((node.right, height + 1))
31
32
33 if __name__ == '__main__':
34     root = Node(1)
35     root.left = Node(2)
36     root.right = Node(3)
37     root.left.left = Node(4)
38     root.left.right = Node(5)
39
40     print(get_min_depth(root))

```

Remove nodes on root to leaf paths of length < K

```

1  # Remove nodes on root to leaf paths of length < K
2
3
4  class Node:
5      def __init__(self, data):
6          self.data = data
7          self.left = None
8          self.right = None
9
10
11 def remove_path_less_than(root, k):
12     return remove_path_util(root, 1, k)
13
14
15 def remove_path_util(root, level, k):
16     if root is None:

```

```
17         return None
18
19     root.left = remove_path_util(root.left, level+1, k)
20     root.right = remove_path_util(root.right, level+1, k)
21
22     if root.left is None and root.right is None and level < k:
23         del root
24         return None
25
26     return root
27
28
29 def in_order(root):
30     if root is None:
31         return
32     in_order(root.left)
33     print(root.data, end=' ')
34     in_order(root.right)
35
36
37 if __name__ == '__main__':
38     root = Node(1)
39     root.left = Node(2)
40     root.right = Node(3)
41     root.left.left = Node(4)
42     root.left.right = Node(5)
43     root.left.left.left = Node(7)
44     root.right.right = Node(6)
45     root.right.right.left = Node(8)
46     in_order(root)
47     print()
48     root = remove_path_less_than(root, 4)
49     in_order(root)
```

Given a Perfect Binary Tree, reverse the alternate level nodes of the tree

```
1  # Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree
2
3
4  class Node:
5      def __init__(self, data):
6          self.data = data
7          self.left = None
8          self.right = None
9
10
11 def reverse_alt_levels(root):
12     pre_order_rev(root.left, root.right, 0)
13
14
15 def pre_order_rev(root_left, root_right, level):
16     # Base case
17     if (root_left or root_right) is None:
18         return
19
20     # swap the data of nodes if at an alternate level
21     if level % 2 == 0:
22         root_left.data, root_right.data = root_right.data, root_left.data
23
24     # go to the next level with left of left root and right of right root
25     # and vice versa
26     pre_order_rev(root_left.left, root_right.right, level+1)
27     pre_order_rev(root_left.right, root_right.left, level+1)
28
29
30 def in_order(root):
31     if root is None:
32         return
33     in_order(root.left)
34     print(root.data, end=' -> ')
35     in_order(root.right)
36
37
38 if __name__ == '__main__':
39     root = Node('a')
40     root.left = Node('b')
41     root.right = Node('c')
42     root.left.left = Node('d')
43     root.left.right = Node('e')
```



```

44     root.right.left = Node('f')
45     root.right.right = Node('g')
46     root.left.left.left = Node('h')
47     root.left.left.right = Node('i')
48     root.left.right.left = Node('j')
49     root.left.right.right = Node('k')
50     root.right.left.left = Node('l')
51     root.right.left.right = Node('m')
52     root.right.right.left = Node('n')
53     root.right.right.right = Node('o')
54
55     print('Before Reversal:')
56     in_order(root)
57     print()
58
59     # Call the reverse alternate levels function
60     reverse_alt_levels(root)
61
62     print('After Reversal:')
63     in_order(root)
64     print()

```

Print Nodes in Top View of Binary Tree

```

1  # Print Nodes in Top View of Binary Tree
2  from collections import deque
3
4
5  class Node:
6      def __init__(self, data):
7          self.data = data
8          self.left = None
9          self.right = None
10
11
12  def top_view(root):
13      if root is None:
14          return
15
16      # make an empty queue for BFS
17      q = deque()
18

```

```

19     # empty set
20     sets = set({})
21
22     # list to store top view keys
23     topview = []
24
25     # append root in the queue with horizontal distance as 0
26     q.append((root, 0))
27
28     while q:
29         # get the element and horizontal distance
30         elem, hd = q.popleft()
31
32         # if the hd is seen first time it will be top view
33         if hd not in sets:
34             topview.append((elem.data, hd))
35             sets.add(hd)
36
37         # add left and right child in the queue with hd - 1 and hd + 1
38         if elem.left is not None:
39             q.append((elem.left, hd - 1))
40         if elem.right is not None:
41             q.append((elem.right, hd + 1))
42
43     # return the sorted topview on the basis of hd
44     return sorted(topview, key=lambda x: x[1])
45
46
47 if __name__ == '__main__':
48     root = Node(1)
49     root.left = Node(2)
50     root.right = Node(3)
51     root.left.right = Node(4)
52     root.left.right.right = Node(5)
53     root.left.right.right.right = Node(6)
54
55     for i in top_view(root):
56         print(i[0], end=' ')

```

Implementation of Trie data structure

```

1  class Node:
2      def __init__(self, value=None, isComplete=False):
3          self.isComplete = isComplete
4          self.children = {}
5          self.value = value
6          self.isPrefixOf = 0
7
8
9  class Trie:
10     def __init__(self):
11         self.root = Node()
12
13     def add_word(self, word):
14         """
15         Add the given word into the trie
16         :param word: A String (word) to be added in the trie
17         """
18         chars = list(word)
19
20         curr_node = self.root
21
22         for ch in chars:
23             # The substring till this node will now become a prefix of newly added w\
24     ord
25         curr_node.isPrefixOf += 1
26
27         if ch in curr_node.children:
28             curr_node = curr_node.children[ch]
29         else:
30             new_node = Node(value=ch)
31             curr_node.children[ch] = new_node
32             curr_node = new_node
33
34         curr_node.isComplete = True
35
36     def search(self, word):
37         """
38         Searches if the word is present in the Trie or not
39         :param word: String (word) to be searched in the trie
40         :return: last Node of the searched word if present else None
41         """
42         chars = list(word)
43

```

```

44         curr_node = self.root
45
46         for ch in chars:
47             if ch in curr_node.children:
48                 curr_node = curr_node.children[ch]
49             else:
50                 return None
51
52         if curr_node.isComplete is True:
53             return curr_node
54
55         return None
56
57     def delete(self, word):
58         """
59         Deletes the given String (word) from the trie
60         :param word: Word (String) to be deleted
61         :return: True is deleted, False if word not present in the Trie
62         """
63         chars = list(word)
64         n = len(chars)
65
66         val = self._delete(self.root, word)
67         return True if val == 1 or val == 0 else False
68
69     def _delete(self, node, chars):
70         """
71         Recursive Helper function to delete the word and decrement the isPrefix of \
72 values
73         :param node: current node looking at
74         :param chars: array of characters to look for
75         :return: 1 is word is deleted, 0 if word is deleted and
76         """
77
78         # if the chars array is empty
79         if len(chars) == 0:
80             # check if the word is present in the trie
81             if node.isComplete:
82                 node.isComplete = False
83
84             # check if the word was a prefix of any other words in trie
85             # if so, decrement isPrefixOf and return 0, as no deletions are requ\
86             ired

```

```

87         if len(node.children.keys()) > 0:
88             node.isPrefixOf -= 1
89             return 0
90
91         # if word was not a prefix then we need to go up in the trie
92         # and find the lowest parent which forms a new word in trie
93         return 1
94         # if word is not present in the trie
95         return -1
96
97     # check if the character is present in current node's children
98     if chars[0] in node.children:
99         # recursive call for remaining characters in the respective child
100        val = self._delete(node.children[chars[0]], chars[1:])
101
102        # if word was found but lowest parent which forms new word is not found
103        if val == 1:
104            if node.isComplete or len(node.children.keys()) > 1:
105                del node.children[chars[0]]
106                node.isPrefixOf -= 1
107                val = 0
108            # if word was found and lowest parent which forms new word was also found
109            # simply reduce the isPrefixOf value of the node
110            elif val == 0:
111                node.isPrefixOf -= 1
112            return val
113
114        return -1
115
116
117 trie = Trie()
118 trie.add_word("anubhav")
119 trie.add_word("anubshrimal")
120 trie.add_word("anubhavshrimal")
121 trie.add_word("data_structures")
122
123 if trie.search("anubhav") is not None:
124     print("anubhav is present in the Trie")
125 else:
126     print("anubhav is NOT present in the Trie")
127
128 trie.delete("anubhav")
129

```

```
130 if trie.search("anubhav") is not None:
131     print("anubhav is present in the Trie")
132 else:
133     print("anubhav is NOT present in the Trie")
134
135 print("Number of words in trie:", trie.root.isPrefixOf)
```

Keep developing your programming skills

This book is just a starting point for you to get inspired to keep discovering different ways of solving common problems through algorithms.

Hopefully you are now more familiar with some of the common Data Structures and Algorithms, keep practicing your coding skills.

I want to say thanks for taking your time and reading the whole book, I really appreciate your commitment towards your education and professional success, and for that I'd like to point you in the right direction with the following incredible resources which will help you achieve your goals.

- Online Courses with discounts site: [Online Courses](#)⁸
- Facebook community for developers: [The Programming Hub](#)⁹
- Quora Group: [The Programming Hub](#)¹⁰

I hope this book was useful, please feel free to contact me with any questions. Also I would appreciate your honest review or any feedback to improve the next editions of the book.

Thanks a lot for following along !!

⁸<https://coursesim.com>

⁹<https://www.facebook.com/theproghub>

¹⁰https://www.quora.com/q/vlyubtzhdhpacqzr?invite_code=Yj9D6Z7H2Bf4CONidZmD

About the Author

Alejandro is a programmer and writer with the goal of inspiring people to learn about several tech related topics.

He's also a Udemy Instructor here's a link to his profile: [Alejandro Garcia Udemy Instructor](#)¹¹

Other Author Profiles

Amazon: [Alejandro on Amazon](#)¹²

Leanpub: [Alejandro on Leanpub](#)¹³

Copyright

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This book contains some affiliate links, this means, if you purchase anything from those links there won't be an extra charge for you. In addition you will be supporting the future updates of this publication

Some Chapters of the book are based on the following repositories:

Data Structures and Algorithms [MIT License](#)¹⁴

¹¹<https://www.udemy.com/user/alejandro-garcia-172>

¹²<https://www.amazon.com/Alejandro-Garcia/e/B08CF2H6TB>

¹³<https://leanpub.com/u/alejandro-garcia>

¹⁴<https://github.com/anubhavshrimal/Data-Structures-Algorithms/blob/master/LICENSE>