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

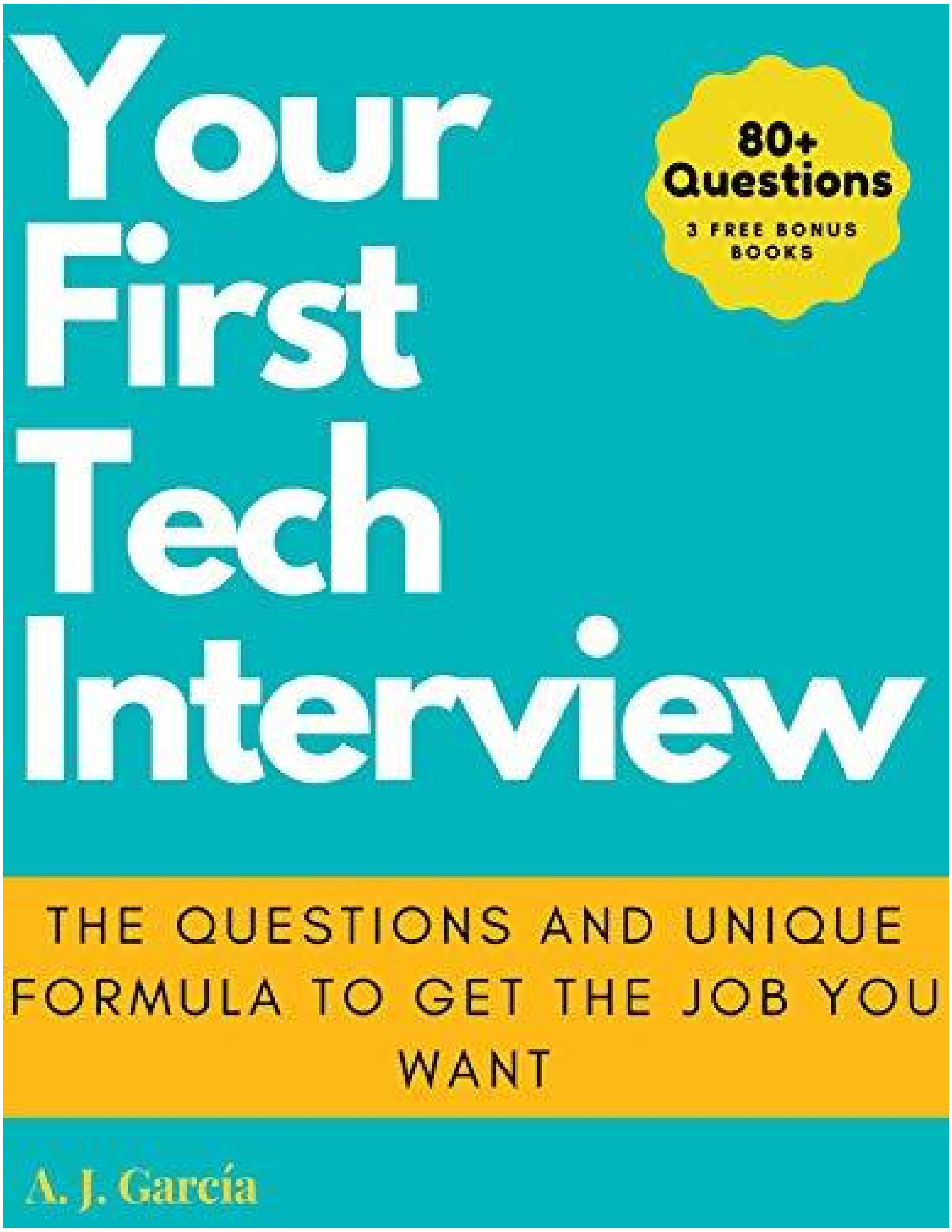
This book is for sale at http://leanpub.com/data-structures-algorithms-for-job-interviews

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.

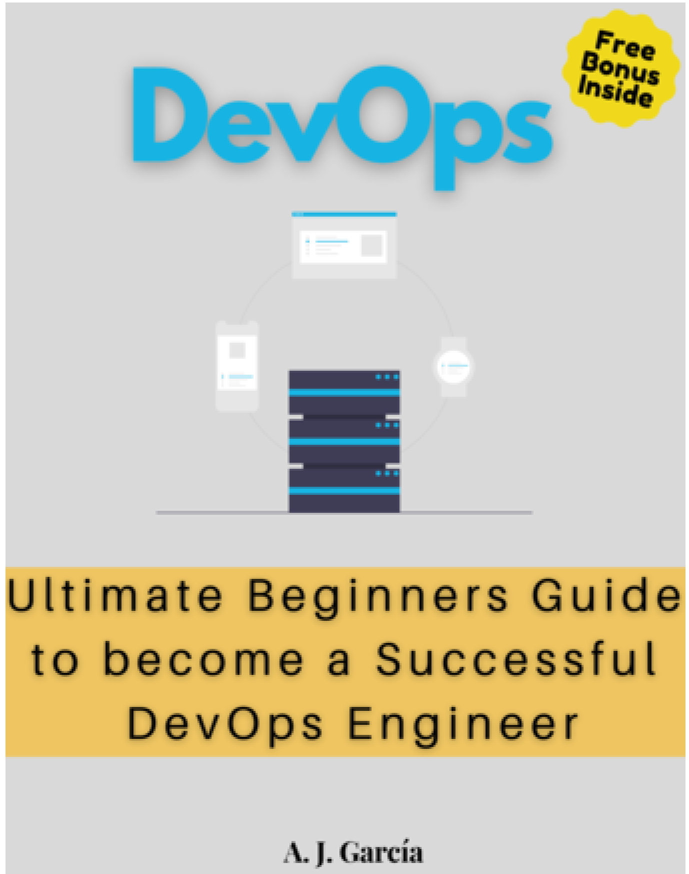
Contents

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 want[1]



DevOps: Ultimate Beginners Guide to become a Successful Devops Engineer

DevOps: Ultimate Beginners Guide to become a Successful Devops Engineer[2]

---

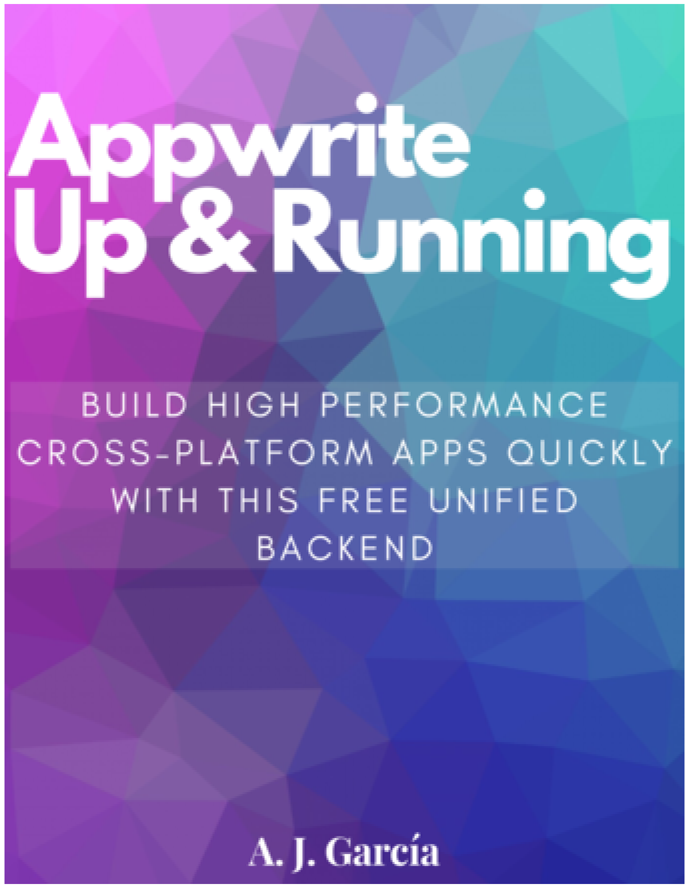
[1]https://www.amazon.com/Your-First-Tech-Interview-Questions-ebook/dp/B08DN1BX2V/ref=sr_1_3?crid=2WMGG8O7DREFP&dchild=1&keywords=tech+interview+questions&qid=1596198458&s=books&sprefix=tech+intervi%2Caps%2C261&sr=1-3

[2]https://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 Apps[3]

[3]https://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 Backend[4]

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]

[4]https://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[7]

[7] https://alejandro-garcia-author.ck.page/2ed39fffaf

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

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


def check_pow_2(num):
    if num == 0:
        return 0

    if num & (num - 1) == 0:
        return 1

    return -1


switch = {
    0: "Number is 0",
    1: "Number is a power of 2",
    -1: "Number is neither a power of 2 nor 0"
}
case = check_pow_2(16)

print(switch[case])
```

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

```python
# Count number of bits needed to be flipped to convert A to B


def count_bits_flip(a, b):
    # XOR a and b to get 1 on opposite value bit position
    c = a ^ b

    # initialise the counter for 1
    count = 0

    # count the number of 1s while there is 1 in a ^ b
    while c != 0:
        count += 1
        c &= (c-1)

    # return the count of 1s
    return count


# 2 = 0010
# 8 = 1000
print(count_bits_flip(2, 8))
```

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

```python
# Find the two non-repeating elements in an array of repeating elements


def find_non_repeating_numbers(arr):
    xor = arr[0]

    for i in range(1, len(arr)):
        xor ^= arr[i]

    right_set_bit = xor & ~(xor-1)
    first = 0
    second = 0
    for i in arr:
        if i & right_set_bit:
            first ^= i
```

```python
        else:
            second ^= i

    return first, second


arr = [2, 3, 7, 9, 11, 2, 3, 11]
print(find_non_repeating_numbers(arr))
```

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

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

# Ex. num = 6 bin = 110
def next_greater(num):
    res = num
    if num != 0:
        # Find the right most 1 position
        # Ex. right_one = 2 bin = 10
        right_one = num & -num

        # get the left pattern to merge
        # Ex. left_pattern = 8 bin = 1000
        left_pattern = num + right_one

        # get the right patten to merge
        # Ex. right_pattern = 1 bin = 0001
        right_pattern = (num ^ left_pattern) >> (right_one + 1)

        # OR both the patterns
        # Ex. res = 9 bin = 1001
        res = left_pattern | right_pattern

    return res


def next_smaller(num):
    return ~next_greater(~num)


```

```
print(next_greater(6))
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 donâ€™t pick it (0-1 property).

```python
def knapSack(W, wt, val, size):
    k = [[0 for i in range(W+1)] for i in range(size+1)]
    for i in range(size+1):
        for w in range(W+1):
            if i == 0 or w == 0:
                k[i][w] = 0
            elif wt[i-1] <= w:
                k[i][w] = max(val[i-1] + k[i-1][w-wt[i-1]], k[i-1][w])
            else:
                k[i][w] = k[i-1][w]

    for w in k:
        print(w)

    return k


# def findElementsInSack(W, matrix, wt, val, size):
#     i = size
#     row = W
#     arr = []
#     while i > 0:
#         print(matrix[i][row] - matrix[i-1][row - wt[i-1]] )
#         print(val[i-1])
#         if matrix[i][row] - matrix[i-1][row - wt[i-1]] == val[i-1]:
#             arr.append(val[i-1])
#             i -= 1
```

```
#             row -= wt[i-1]
#         else:
#             i -= 1
#
#     return arr

price = [60, 100, 120]
wt = [1, 2, 3]
W = 5
n = len(price)
k = knapSack(W, wt, price, n)
print(k[n][W])
# 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.

```
def cutting_rod(prices, n):
    dp = [0 for i in range(n+1)]
    dp[0] = 0

    for i in range(1, n+1):
        max_val = -float('inf')
        for j in range(i):
            max_val = max(max_val, prices[j] + dp[i-j-1])
        dp[i] = max_val

    return dp[n]


if __name__ == '__main__':
    arr = [1, 5, 8, 9, 10, 17, 17, 20]
    size = len(arr)
    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 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

```
def edit_distance(str1, str2, m, n):
    matrix = [[0 for i in range(n+1)] for i in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):

            if i == 0:
                matrix[i][j] = j

            elif j == 0:
                matrix[i][j] = i

            elif str1[i-1] == str2[j-1]:
                matrix[i][j] = matrix[i-1][j-1]

            else:
                matrix[i][j] = 1 + min(matrix[i][j-1],      # insert
                                       matrix[i-1][j],      # remove
                                       matrix[i-1][j-1])    # replace

    return matrix[m][n]


if __name__ == '__main__':
```

```
    str1 = 'sunday'
    str2 = 'saturday'

    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.

```
def find_largest_square(matrix):
    n = len(matrix)

    # make a matrix for storing the solutions
    cache = [[0] * n for _ in range(n)]
    # size of square and its bottom-right indexes
    size = 0
    right_indx = -1
    bottom_indx = -1

    for i in range(n):
        for j in range(n):

            # if the value is 0 simply move forward as it cannot form a square of 1s
            if matrix[i][j] == 0:
                continue

            # if it is first row or column, copy the matrix values as it is
            elif i == 0 or j == 0:
                cache[i][j] = matrix[i][j]

            # Otherwise, check in the up, left, and diagonally top-left direction fo\
r minimum size of square
            # if all are 1s at these positions in matrix, only then min value will b\
e greater than 1
            # hence add the previous square size to the cache + 1
            else:
                cache[i][j] = 1 + min(cache[i - 1][j], cache[i][j - 1], cache[i - 1]\
[j - 1])

            # check if the current square size found is larger than the previously f\
```

```
ound size, if so, update it
            if cache[i][j] > size:
                size = cache[i][j]
                bottom_indx, right_indx = i, j

    return size, bottom_indx, right_indx


mat = [[1, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1]]
size, bottom, right = find_largest_square(mat)

if size > 0:
    print("Size of the square:", size)
    print("Top-left Co-ordinates:", bottom-size+1, ",", right-size+1)
    print("Bottom-right Co-ordinates:", bottom, ",", right)
else:
    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.

```
def lcs(str1, str2):
    m = len(str1)
    n = len(str2)

    matrix = [[0 for i in range(n+1)] for i in range(m+1)]

    for i in range(1, m+1):
        for j in range(1, n+1):

            if i == 0 or j == 0:
```

```
                matrix[i][j] = 0

            elif str1[i-1] == str2[j-1]:
                matrix[i][j] = 1 + matrix[i-1][j-1]

            else:
                matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])

    index = matrix[m][n]

    res = [""] * index
    i = m
    j = n

    while i > 0 and j > 0:
        if str1[i-1] == str2[j-1]:
            res[index-1] = str1[i-1]
            i -= 1
            j -= 1
            index -= 1

        elif matrix[i-1][j] > matrix[i][j-1]:
            i -= 1
        else:
            j -= 1

    return res


if __name__ == '__main__':
    X = "AGGTAB"
    Y = "GXTXAYB"

    str = ''.join(lcs(X, Y))

    print("Length of longest common subsequence is:", len(str),"\nAnd the subsequenc\
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.

```
def lis(arr):
    n = len(arr)
    dp = [1] * n

    for i in range(n):
        for j in range(i):
            if arr[j] < arr[i] and dp[j] + 1 > dp[i]:
                dp[i] = 1 + dp[j]

    return max(dp)


arr = [10, 22, 9, 33, 21, 50, 41, 60, 80]
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.

```python
def min_cost(cost, m, n):
    dp = [[0 for i in range(n+1)] for i in range(m+1)]

    dp[0][0] = cost[0][0]

    for i in range(1, m+1):
        dp[i][0] = dp[i-1][0] + cost[i][0]

    for j in range(1, n+1):
        dp[0][j] = dp[0][j-1] + cost[0][j]

    for i in range(1, m+1):
        for j in range(1, n+1):
            dp[i][j] = cost[i][j] + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

    return dp[m][n]


if __name__ == '__main__':
    cost = [[1, 2, 3],
            [4, 8, 2],
            [1, 5, 3]]
    m = 2
    n = 2
    print("Minimum cost from (0, 0) to ({}, {}) is:".format(m, n), min_cost(cost, m,\
 n))
```

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

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


def find_min(arr):
    sum_of_arr = sum(arr)
    n = len(arr)
    dp = [[False for i in range(sum_of_arr+1)] for i in range(n+1)]

    for i in range(n+1):
        dp[i][0] = True

```

```
    for i in range(1, sum_of_arr+1):
        dp[0][i] = False

    for i in range(1, n+1):
        for j in range(1, sum_of_arr+1):
            dp[i][j] = dp[i-1][j]

            if arr[i-1] <= j:
                dp[i][j] |= dp[i-1][j - arr[i-1]]

    diff = float('inf')

    for j in range(int(sum_of_arr/2), -1, -1):
        if dp[n][j] is True:
            diff = sum_of_arr - 2 * j
            break

    return diff


if __name__ == '__main__':
    arr = [3, 1, 4, 2, 2, 1]
    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 mi in M denotes the exact number of people (mi), the ith 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.

```
def min_umbrellas_needed_util(n, umbrellas, dp, count):
    # if dp has already stored the solution for n people, return
    if n == 0 or dp[n] != 0:
        return dp[n]

    min_count = float('inf')
    curr_count = None

    # Iterate over all the umbrella sizes
```

```
    for m in umbrellas:
        # if umbrella can be accomodated fully
        if n - m > 0:
            curr_count = min_umbrellas_needed_util(n-m, umbrellas, dp, count+1)

        # if all people are accomodated
        elif n - m == 0:
            curr_count = count+1

        # if the umbrella cannot get exact number of people to fit
        else:
            curr_count = float('inf')

        if curr_count < min_count:
            min_count = curr_count

    # memoize result
    dp[n] = min_count

    return min_count


def min_umbrellas_needed(n, umbrellas):
    # initialize a dp table for memoization
    dp = [0] * (n+1)

    count = min_umbrellas_needed_util(n, umbrellas, dp, 0)

    if count == float('inf'):
        return -1

    return count

umbrellas = [5,4,2,1]
n = 8

print(f"Number of umbrellas needed to accomodate {n} people: {min_umbrellas_needed(n\
, 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.

```python
def isSubsetSum(arr, check_sum):
    n = len(arr)
    possible_sum = [[False] * (n + 1) for _ in range(check_sum + 1)]

    for i in range(n+1):
        possible_sum[0][i] = True

    for i in range(1, check_sum + 1):
        for j in range(1, n + 1):
            if i < arr[j - 1]:
                possible_sum[i][j] = possible_sum[i][j-1]
            elif i >= arr[j - 1]:
                possible_sum[i][j] = possible_sum[i][j-1] or possible_sum[i - arr[j \
- 1]][j-1]

    return possible_sum[check_sum][n]


arr = [3, 34, 4, 12, 5, 2]
check_sum = 9

if isSubsetSum(arr, check_sum):
    print("Found a subset with sum =", check_sum)
else:
    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

```
def count_ways(n):
    count = [0] * (n+1)
    count[0] = 1
    count[1] = 1
    count[2] = 2

    for i in range(3, n+1):
        count[i] = count[i-1] + count[i-2] + count[i-3]

    return count[n]


if __name__ == '__main__':
    print(count_ways(4))
```

Chapter 3: Graph

Find all possible words in a board of characters

```python
# Find all possible words in a board of characters
"""Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                       {'U','E','K'},
                       {'Q','S','E'}};
Output:  Following words of dictionary are present
         GEEKS
         QUIZ
"""


def findWordsUtil(words, boggle, visited, found, r, c, str):
    rows = len(boggle)
    cols = len(boggle[0])

    # set the position of character as traversed
    visited[r][c] = True

    # add the character to string
    str += boggle[r][c]

    # if the string is in dictionary add it to the set of found words
    if str in words:
        found.add(str)

    # traverse all the nearby 8 adjacent cells
    for i in range(r-1, r+2):
        for j in range(c-1, c+2):
            if i >= rows or i < 0 or j >= cols or j < 0 or visited[i][j]:
                continue
            findWordsUtil(words, boggle, visited, found, i, j, str)

    # backtrack and set the status of current character as not traversed
    visited[r][c] = False

```

```python

def findWords(words, boggle):
    rows = len(boggle)
    cols = len(boggle[0])

    # initialize a matrix for DFS Traversal
    visited = [[False for i in range(cols)] for j in range(rows)]

    # set to store the unique found words
    found = set({})
    str = ""

    # traverse each character in the boggle and do DFS from there
    for r in range(rows):
        for c in range(cols):
            findWordsUtil(words, boggle, visited, found, r, c, str)

    # return the set of found words
    return found

if __name__ == '__main__':
    words = {"GEEKS", "FOR", "QUIZ", "GO", "SEEK"}
    boggle = [['G', 'I', 'Z'],
              ['U', 'E', 'K'],
              ['Q', 'S', 'E']]

    found = findWords(words, boggle)

    print("Words found in the boggle from the dictionary are:")
    for word in found:
        print(word)
```

Breadth First Search Traversal

```
# Program to perform breadth first traversal in a graph
from collections import defaultdict, deque


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to):
        self.graph[frm].append(to)

        if self.directed is False:
            self.graph[to].append(frm)
        else:
            self.graph[to] = self.graph[to]

    def bfsUtil(self, s, visited):
        queue = deque([])
        queue.append(s)
        visited[s] = True

        while queue:
            vertex = queue.popleft()
            print(vertex, end=' ')

            # traverse vertices adjacent to vertex
            for i in self.graph[vertex]:
                if not visited[i]:
                    visited[i] = True
                    queue.append(i)
        print()

    def bfs(self, s=None):
        visited = {i: False for i in self.graph}

        # do bfs from the node specified
        if s is not None:
            self.bfsUtil(s, visited)
        # traverse for all the vertices in other components of graph
        for v in self.graph:
            if not visited[v]:
                self.bfsUtil(v, visited)
```

```python

if __name__ == '__main__':
    graph = Graph()

    # component 1 of the graph
    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(1, 2)
    graph.addEdge(2, 3)
    graph.addEdge(3, 3)
    graph.addEdge(1, 4)
    graph.addEdge(1, 5)
    graph.addEdge(3, 6)

    # component 2 of the graph
    graph.addEdge(7, 8)
    graph.addEdge(8, 9)
    graph.addEdge(7, 10)

    # call bfs from 2 vertex
    print("Breadth First Traversal:")
    graph.bfs(2)
```

Depth First Search Traversal

```python
# Program to perform depth first traversal in a graph
from collections import defaultdict


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to):
        self.graph[frm].append(to)

        if self.directed is False:
            self.graph[to].append(frm)
        else:
            self.graph[to] = self.graph[to]

```

```python
    def dfsUtil(self, s, visited):
        stack = []
        stack.append(s)
        visited[s] = True

        while stack:
            vertex = stack.pop()
            print(vertex, end=' ')

            # traverse vertices adjacent to vertex
            for i in self.graph[vertex]:
                if not visited[i]:
                    visited[i] = True
                    stack.append(i)
        print()

    def dfs(self, s=None):
        visited = {i: False for i in self.graph}

        # traverse specified vertex
        if s is not None:
            self.dfsUtil(s, visited)

        # traverse for all the vertices in other components of graph
        for v in self.graph:
            if not visited[v]:
                self.dfsUtil(v, visited)


if __name__ == '__main__':
    # make an undirected graph
    graph = Graph()

    # component 1 of the graph
    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(1, 2)
    graph.addEdge(2, 3)
    graph.addEdge(3, 3)
    graph.addEdge(1, 4)
    graph.addEdge(1, 5)
    graph.addEdge(3, 6)

```

```
    # component 2 of the graph
    graph.addEdge(7, 8)
    graph.addEdge(8, 9)
    graph.addEdge(7, 10)

    # call dfs from 2 vertex
    print("Depth First Traversal:")
    graph.dfs(2)
```

Detect Cycle in directed graph

```
# Program to detect cycle or loop in a directed graph
from collections import defaultdict


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to):
        self.graph[frm].append(to)

        if self.directed is False:
            self.graph[to].append(frm)
        else:
            self.graph[to] = self.graph[to]

    def isCyclicUtil(self, s, visited, recurStack):

        if visited[s] is False:
            recurStack[s] = True
            visited[s] = True

            # traverse vertices adjacent to vertex
            for i in self.graph[s]:
                if (not visited[i]) and self.isCyclicUtil(i, visited, recurStack):
                    return True
                elif recurStack[i]:
                    return True
        recurStack[s] = False
        return False
```

```python

    def isCyclic(self):
        visited = {i: False for i in self.graph}
        recurStack = {i: False for i in self.graph}

        # traverse for all the vertices of graph
        for v in self.graph:
            if self.isCyclicUtil(v, visited, recurStack):
                return True
        return False


if __name__ == '__main__':
    # make a directed graph
    graph = Graph(True)

    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(1, 2)
    graph.addEdge(2, 0)
    graph.addEdge(2, 3)
    graph.addEdge(3, 3)

    if graph.isCyclic():
        print("Cycle exists")
    else:
        print("No cycle in the graph")
```

Detect cycle in undirected graph

```python
# Program to detect cycle or loop in a graph
from collections import defaultdict


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to):
        # True if edge has been traversed or seen once
        self.graph[frm].append([to, False])
```

```

        if self.directed is False:
            self.graph[to].append([frm, False])
        else:
            self.graph[to] = self.graph[to]

    def findParent(self, sets, v):
        if sets[v] == -1:
            return v
        else:
            return self.findParent(sets, sets[v])

    def union(self, sets, x, y):
        x_set = self.findParent(sets, x)
        y_set = self.findParent(sets, y)
        sets[x_set] = y_set

    def isCyclic(self):
        # sets that show combined vertices or not
        sets = {i: -1 for i in self.graph}

        for v in self.graph:
            for e in self.graph[v]:
                # if an edge is traversed once skip it
                if e[1] is True:
                    continue

                # set True for traversing the edge and making union in both adjacenc\
y lists
                e[1] = True

                for i in self.graph[e[0]]:
                    if i[0] == v:
                        i[1] = True
                        break

                # find parents of both vertices of the edge
                x = self.findParent(sets, v)
                y = self.findParent(sets, e[0])

                # if they share a common parent loop found
                if x == y:
                    return True
```

```
                # union the two vertices in the same set
                self.union(sets, x, y)

        # if no loop or cycle found return false
        return False


if __name__ == '__main__':
    # make a graph
    graph = Graph()
    graph.addEdge(0, 1)
    graph.addEdge(1, 2)
    graph.addEdge(2, 0)

    if graph.isCyclic():
        print("Cycle exists in the graph")
    else:
        print("No cycle in the graph")
```

Dijkstra’s Shortest Path Algorithm

```
# Given a graph and a source vertex in graph, find shortest paths from source to all\
 vertices in the given graph
from collections import defaultdict


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to, weight):
        self.graph[frm].append([to, weight])

        if self.directed is False:
            self.graph[to].append([frm, weight])
        else:
            self.graph[to] = self.graph[to]

    def find_min(self, dist, visited):
        minimum = float('inf')
        index = -1
```

```
        for v in self.graph.keys():
            if visited[v] is False and dist[v] < minimum:
                minimum = dist[v]
                index = v

        return index

    def dijkstra(self, src):
        visited = {i: False for i in self.graph}
        dist = {i: float('inf') for i in self.graph}
        parent = {i: None for i in self.graph}

        # set distance of src vertex from itself 0
        dist[src] = 0

        # find shortest path for all vertices
        for i in range(len(self.graph)-1):
            # find minimum distance vertex from source
            # initially src itself as dist[src] = 0
            u = self.find_min(dist, visited)

            # mark the node as visited
            visited[u] = True
            # check if the distance through current edge is less than previously kno\
wn distance to v
            for v, w in self.graph[u]:

                if visited[v] is False and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u
        # return parent list and distance to each node from source
        return parent, dist

    def printPath(self, parent, v):
        if parent[v] is None:
            return
        self.printPath(parent, parent[v])
        print(v, end=' ')

    def printSolution(self, dist, parent, src):
        print('{}\t{}\t{}'.format('Vertex', 'Distance', 'Path'))

        for i in self.graph.keys():
```

```
            if i == src:
                continue
            print('{} -> {}\t\t{}\t\t{}'.format(src, i, dist[i], src), end=' ')
            self.printPath(parent, i)
            print()

if __name__ == '__main__':
    # make an undirected graph
    graph = Graph()

    graph.addEdge(0, 1, 4)
    graph.addEdge(0, 7, 8)
    graph.addEdge(1, 2, 8)
    graph.addEdge(1, 7, 11)
    graph.addEdge(7, 6, 1)
    graph.addEdge(7, 8, 7)
    graph.addEdge(6, 8, 6)
    graph.addEdge(6, 5, 2)
    graph.addEdge(8, 2, 2)
    graph.addEdge(2, 3, 7)
    graph.addEdge(2, 5, 4)
    graph.addEdge(3, 4, 9)
    graph.addEdge(3, 5, 14)
    graph.addEdge(5, 4, 10)

    parent, dist = graph.dijkstra(0)

    graph.printSolution(dist, parent, 0)
```

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

```
# Find shortest distances between every pair of vertices in a given edge weighted di\
rected Graph
"""
              10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
             3
"""


def floyd_warshall(graph):
    shortest_dist = []

    # copy matrix for storing resultant shortest distances
    for i in graph:
        shortest_dist.append(i)

    # Number of vertices in graph
    V = len(graph) - 1

    # k is intermediate vertex
    for k in range(V+1):
        # i is source
        for i in range(V+1):
            # j is destination
            for j in range(V+1):
                # store the path which is shorter i.e. min(i->j, i->k->j)
                shortest_dist[i][j] = min(shortest_dist[i][j], shortest_dist[i][k] +\
 shortest_dist[k][j])
    # return the resultant matrix
    return shortest_dist

if __name__ == '__main__':
    INF = float('inf')
    graph = [[0, 5, INF, 10],
             [INF, 0, 3, INF],
             [INF, INF, 0, 1],
             [INF, INF, INF, 0]]

```

```python
    shortest_dist_matrix = floyd_warshall(graph)

    for i in shortest_dist_matrix:
        for j in i:
            if j != float('inf'):
                print(j, '\t', end='')
            else:
                print(j, end=' ')
        print()
```

Graph implementation

```python
class Vertex:
    def __init__(self, key):
        self.key = key
        self.adjacent = {}
        self.visited = False

    def setKey(self, key):
        self.key = key

    def getKey(self):
        return self.key

    def getVisited(self):
        return self.visited

    def setVisited(self, val=True):
        self.visited = val

    def addNeighbour(self, neighbour, weight=0):
        self.adjacent[neighbour] = weight

    def getNeighbours(self):
        return self.adjacent.keys()

    def getWeight(self, neighbour):
        return self.adjacent[neighbour]


class Graph:

```

```python
    # Graph is undirected by default
    def __init__(self, directed=False):
        self.vertices = {}
        self.numberOfVertices = 0
        self.directed = directed

    def addVertex(self, key):
        node = Vertex(key)
        self.vertices[key] = node
        self.numberOfVertices += 1
        return node

    def addEdge(self, frm, to, weight=0):
        if frm not in self.vertices:
            self.addVertex(frm)

        if to not in self.vertices:
            self.addVertex(to)

        self.vertices[frm].addNeighbour(self.vertices[to], weight)

        if not self.directed:
            self.vertices[to].addNeighbour(self.vertices[frm], weight)

    def getVertex(self, key):
        if key in self.vertices:
            return self.vertices[key]
        else:
            return None

    def getVertices(self):
        return self.vertices.keys()

    def getEdges(self):
        edges = []
        for v in self.vertices:
            edgesFromVertex = []

            for w in self.vertices[v].getNeighbours():
                frm = self.vertices[v].getKey()
                to = w.getKey()
                weight = self.vertices[v].getWeight(w)
                edgesFromVertex.append((frm, to, weight))
```

```python

            if len(edgesFromVertex) != 0:
                edges.append(edgesFromVertex)

        return edges


if __name__ == '__main__':
    g = Graph(directed=False)
    g.addVertex('a')
    g.addVertex('b')
    g.addVertex('c')
    g.addVertex('d')
    g.addVertex('e')
    g.addVertex('f')
    g.addEdge('a', 'b', 3)
    g.addEdge('b', 'c', 2)
    g.addEdge('c', 'd', 1)
    g.addEdge('d', 'e', 5)
    g.addEdge('d', 'a', 5)
    g.addEdge('d', 'a', 2)
    g.addEdge('e', 'f', 3)
    g.addEdge('f', 'a', 6)
    g.addEdge('f', 'b', 6)
    g.addEdge('f', 'c', 6)

    for edgeSet in g.getEdges():
        print('edges from', edgeSet[0][0] + ': ', end='')
        print(edgeSet)
```

Kruskal’s Algorithm for Minimum Spanning Tree

```python
# Kruskal’s Minimum Spanning Tree Algorithm


class Graph:
    def __init__(self, directed=False):
        self.edges = []
        self.vertices = set({})
        self.directed = directed

    def addEdge(self, frm, to, weight):
        self.edges.append([frm, to, weight])
        self.vertices.add(frm)
        self.vertices.add(to)

    def removeEdge(self, frm, to, weight):
        self.edges.remove([frm, to, weight])
        flag1 = 0
        flag2 = 0
        for f, t, w in self.edges:
            if frm == f or frm == t:
                flag1 = 1
            if to == f or to == t:
                flag2 = 1
            if flag1 == 1 and flag2 == 1:
                break

        if flag1 != 1:
            self.vertices.remove(frm)

        if flag2 != 1:
            self.vertices.remove(to)

    def findParent(self, sets, v):
        if sets[v] == -1:
            return v
        else:
            return self.findParent(sets, sets[v])

    def union(self, sets, x, y):
        x_set = self.findParent(sets, x)
        y_set = self.findParent(sets, y)
        sets[x_set] = y_set
```

```python
    def isCyclic(self):
        # sets that show combined vertices or not
        sets = {i: -1 for i in self.vertices}
        for v1, v2, w in self.edges:
            # find parents of both vertices of the edge
            x = self.findParent(sets, v1)
            y = self.findParent(sets, v2)

            # if they share a common parent loop found
            if x == y:
                return True
            # union the two vertices in the same set
            self.union(sets, x, y)

        # if no loop or cycle found return false
        return False

    def kruskalMST(self):
        g = Graph()

        self.edges = sorted(self.edges, key=lambda x: x[2])

        for frm, to, w in self.edges:
            if len(g.edges) == len(graph.vertices)-1:
                break
            g.addEdge(frm, to, w)
            if g.isCyclic():
                g.removeEdge(frm, to, w)
        return g


if __name__ == '__main__':
    # make an undirected graph
    graph = Graph()

    graph.addEdge(0, 1, 10)
    graph.addEdge(0, 2, 6)
    graph.addEdge(0, 3, 5)
    graph.addEdge(1, 3, 15)
    graph.addEdge(2, 3, 4)

    new_graph = graph.kruskalMST()

```

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

Topological Sort

```python
# Program to perform topological sort in a graph

from collections import defaultdict


class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def addEdge(self, frm, to):
        self.graph[frm].append(to)

        if self.directed is False:
            self.graph[to].append(frm)
        else:
            self.graph[to] = self.graph[to]

    def topoSortUtil(self, s, visited, sortList):
        visited[s] = True

        for i in self.graph[s]:
            if not visited[i]:
                self.topoSortUtil(i, visited, sortList)

        sortList.insert(0, s)

    def topologicalSort(self):
        visited = {i: False for i in self.graph}

        sortList = []
        # traverse for all the vertices in other components of graph
        for v in self.graph:
            if not visited[v]:
                self.topoSortUtil(v, visited, sortList)

        print(sortList)
```

```python


if __name__ == '__main__':
    # make an directed graph
    g = Graph(directed=True)

    g.addEdge(5, 2)
    g.addEdge(5, 0)
    g.addEdge(4, 0)
    g.addEdge(4, 1)
    g.addEdge(2, 3)
    g.addEdge(3, 1)

    # call topologicalSort()
    print("Topological Sort:")
    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

```python
def max_heapify(indx, arr, size):
    """
    Assuming sub trees are already max heaps, converts tree rooted at current indx i\
nto a max heap.
    :param indx: Index to check for max heap
    :param arr: array of elements
    :param size: size of the array
    """

    # Get index of left and right child of indx node
    left_child = indx * 2 + 1
    right_child = indx * 2 + 2

    largest = indx

    # check what is the largest value node in indx, left child and right child
    if left_child < size:
        if arr[left_child] > arr[largest]:
            largest = left_child
    if right_child < size:
        if arr[right_child] > arr[largest]:
            largest = right_child

    # if indx node is not the largest value, swap with the largest child
    # and recursively call max_heapify on the respective child swapped with
    if largest != indx:
        arr[indx], arr[largest] = arr[largest], arr[indx]
        max_heapify(largest, arr, size)


```

```python
def create_max_heap(arr):
    """
    Converts a given array into a max heap
    :param arr: input array of numbers
    :return: output max heap
    """
    n = len(arr)

    # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
    # loop from n/2 to 0 index and convert each index node into max heap
    for i in range(int(n/2), -1, -1):
        max_heapify(i, arr, n)


def heap_sort(arr):
    """
    Sorts the given array using heap sort
    :param arr: input array to sort
    """

    create_max_heap(arr)
    heap_size = len(arr)

    # Swap the max value in heap with the end of the array and decrease the size of \
the heap by 1
    # call max heapify on the 0th index of the array
    while heap_size > 1:
        arr[heap_size-1], arr[0] = arr[0], arr[heap_size-1]
        heap_size -= 1
        max_heapify(0, arr, heap_size)


heap = [5, 10, 4, 8, 3, 0, 9, 11]
heap_sort(heap)
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

```python
from math import ceil


class MaxHeap:
    def __init__(self, arr=None):
        self.heap = []
        self.heap_size = 0
        if arr is not None:
            self.create_max_heap(arr)
            self.heap = arr
            self.heap_size = len(arr)

    def create_max_heap(self, arr):
        """
        Converts a given array into a max heap
        :param arr: input array of numbers
        """
        n = len(arr)

        # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
        # loop from n/2 to 0 index and convert each index node into max heap
        for i in range(int(n / 2), -1, -1):
            self.max_heapify(i, arr, n)

    def max_heapify(self, indx, arr, size):
        """
        Assuming sub trees are already max heaps, converts tree rooted at current in\
dx into a max heap.
        :param indx: Index to check for max heap
        """

        # Get index of left and right child of indx node
        left_child = indx * 2 + 1
        right_child = indx * 2 + 2

        largest = indx

        # check what is the largest value node in indx, left child and right child
        if left_child < size:
            if arr[left_child] > arr[largest]:
                largest = left_child
        if right_child < size:
            if arr[right_child] > arr[largest]:
```

```
                largest = right_child

        # if indx node is not the largest value, swap with the largest child
        # and recursively call min_heapify on the respective child swapped with
        if largest != indx:
            arr[indx], arr[largest] = arr[largest], arr[indx]
            self.max_heapify(largest, arr, size)

    def insert(self, value):
        """
        Inserts an element in the max heap
        :param value: value to be inserted in the heap
        """
        self.heap.append(value)
        self.heap_size += 1

        indx = self.heap_size - 1

        # Get parent index of the current node
        parent = int(ceil(indx / 2 - 1))

        # Check if the parent value is smaller than the newly inserted value
        # if so, then replace the value with the parent value and check with the new\
parent
        while parent >= 0 and self.heap[indx] > self.heap[parent]:
            self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
            indx = parent
            parent = int(ceil(indx / 2 - 1))

    def delete(self, indx):
        """
        Deletes the value on the specified index node
        :param indx: index whose node is to be removed
        :return: Value of the node deleted from the heap
        """
        if self.heap_size == 0:
            print("Heap Underflow!!")
            return

        self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
        self.heap_size -= 1

        self.max_heapify(indx, self.heap, self.heap_size)
```

```python

        return self.heap.pop()

    def extract_max(self):
        """
        Extracts the maximum value from the heap
        :return: extracted max value
        """
        return self.delete(0)

    def print(self):
        print(*self.heap)

heap = MaxHeap([5, 10, 4, 8, 3, 0, 9, 11])

heap.insert(15)
print(heap.delete(2))
print(heap.extract_max())
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

```python
from math import ceil


class MinHeap:
    def __init__(self, arr=None):
        self.heap = []
        self.heap_size = 0
        if arr is not None:
            self.create_min_heap(arr)
            self.heap = arr
            self.heap_size = len(arr)

    def create_min_heap(self, arr):
        """
        Converts a given array into a min heap
        :param arr: input array of numbers
        """
```

```
        n = len(arr)

        # last n/2 elements will be leaf nodes (CBT property) hence already min heaps
        # loop from n/2 to 0 index and convert each index node into min heap
        for i in range(int(n / 2), -1, -1):
            self.min_heapify(i, arr, n)

    def min_heapify(self, indx, arr, size):
        """
        Assuming sub trees are already min heaps, converts tree rooted at current in\
dx into a min heap.
        :param indx: Index to check for min heap
        """
        # Get index of left and right child of indx node
        left_child = indx * 2 + 1
        right_child = indx * 2 + 2

        smallest = indx

        # check what is the smallest value node in indx, left child and right child
        if left_child < size:
            if arr[left_child] < arr[smallest]:
                smallest = left_child
        if right_child < size:
            if arr[right_child] < arr[smallest]:
                smallest = right_child

        # if indx node is not the smallest value, swap with the smallest child
        # and recursively call min_heapify on the respective child swapped with
        if smallest != indx:
            arr[indx], arr[smallest] = arr[smallest], arr[indx]
            self.min_heapify(smallest, arr, size)

    def insert(self, value):
        """
        Inserts an element in the min heap
        :param value: value to be inserted in the heap
        """
        self.heap.append(value)
        self.heap_size += 1

        indx = self.heap_size - 1

```

```
        # Get parent index of the current node
        parent = int(ceil(indx / 2 - 1))

        # Check if the parent value is smaller than the newly inserted value
        # if so, then replace the value with the parent value and check with the new\
 parent
        while parent >= 0 and self.heap[indx] < self.heap[parent]:
            self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
            indx = parent
            parent = int(ceil(indx / 2 - 1))

    def delete(self, indx):
        """
        Deletes the value on the specified index node
        :param indx: index whose node is to be removed
        :return: Value of the node deleted from the heap
        """
        if self.heap_size == 0:
            print("Heap Underflow!!")
            return

        self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
        self.heap_size -= 1

        self.min_heapify(indx, self.heap, self.heap_size)

        return self.heap.pop()

    def extract_min(self):
        """
        Extracts the minimum value from the heap
        :return: extracted min value
        """
        return self.delete(0)

    def print(self):
        print(*self.heap)


heap = MinHeap([5, 10, 4, 8, 3, 0, 9, 11])

heap.insert(15)
print(heap.delete(2))
```

```
print(heap.extract_min())
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, ai, 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

```
from math import ceil


class MaxHeap:
    def __init__(self, arr=None):
        self.heap = []
        self.heap_size = 0
        if arr is not None:
            self.create_max_heap(arr)
            self.heap = arr
            self.heap_size = len(arr)

    def create_max_heap(self, arr):
        """
        Converts a given array into a max heap
        :param arr: input array of numbers
        """
        n = len(arr)

        # last n/2 elements will be leaf nodes (CBT property) hence already max heaps
        # loop from n/2 to 0 index and convert each index node into max heap
        for i in range(int(n / 2), -1, -1):
            self.max_heapify(i, arr, n)

    def max_heapify(self, indx, arr, size):
        """
        Assuming sub trees are already max heaps, converts tree rooted at current in\
dx into a max heap.
        :param indx: Index to check for max heap
        """

        # Get index of left and right child of indx node
        left_child = indx * 2 + 1
        right_child = indx * 2 + 2

        largest = indx

        # check what is the largest value node in indx, left child and right child
        if left_child < size:
            if arr[left_child] > arr[largest]:
                largest = left_child
        if right_child < size:
            if arr[right_child] > arr[largest]:
```

```
                largest = right_child

        # if indx node is not the largest value, swap with the largest child
        # and recursively call min_heapify on the respective child swapped with
        if largest != indx:
            arr[indx], arr[largest] = arr[largest], arr[indx]
            self.max_heapify(largest, arr, size)

    def insert(self, value):
        """
        Inserts an element in the max heap
        :param value: value to be inserted in the heap
        """
        self.heap.append(value)
        self.heap_size += 1

        indx = self.heap_size - 1

        # Get parent index of the current node
        parent = int(ceil(indx / 2 - 1))

        # Check if the parent value is smaller than the newly inserted value
        # if so, then replace the value with the parent value and check with the new\
parent
        while parent >= 0 and self.heap[indx] > self.heap[parent]:
            self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
            indx = parent
            parent = int(ceil(indx / 2 - 1))

    def delete(self, indx):
        """
        Deletes the value on the specified index node
        :param indx: index whose node is to be removed
        :return: Value of the node deleted from the heap
        """
        if self.heap_size == 0:
            print("Heap Underflow!!")
            return

        self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
        self.heap_size -= 1

        self.max_heapify(indx, self.heap, self.heap_size)
```

```

        return self.heap.pop()

    def extract_max(self):
        """
        Extracts the maximum value from the heap
        :return: extracted max value
        """
        return self.delete(0)

    def max(self):
        return self.heap[0]


class MinHeap:
    def __init__(self, arr=None):
        self.heap = []
        self.heap_size = 0
        if arr is not None:
            self.create_min_heap(arr)
            self.heap = arr
            self.heap_size = len(arr)

    def create_min_heap(self, arr):
        """
        Converts a given array into a min heap
        :param arr: input array of numbers
        """
        n = len(arr)

        # last n/2 elements will be leaf nodes (CBT property) hence already min heaps
        # loop from n/2 to 0 index and convert each index node into min heap
        for i in range(int(n / 2), -1, -1):
            self.min_heapify(i, arr, n)

    def min_heapify(self, indx, arr, size):
        """
        Assuming sub trees are already min heaps, converts tree rooted at current in\
dx into a min heap.
        :param indx: Index to check for min heap
        """
        # Get index of left and right child of indx node
        left_child = indx * 2 + 1
```

```python
        right_child = indx * 2 + 2

        smallest = indx

        # check what is the smallest value node in indx, left child and right child
        if left_child < size:
            if arr[left_child] < arr[smallest]:
                smallest = left_child
        if right_child < size:
            if arr[right_child] < arr[smallest]:
                smallest = right_child

        # if indx node is not the smallest value, swap with the smallest child
        # and recursively call min_heapify on the respective child swapped with
        if smallest != indx:
            arr[indx], arr[smallest] = arr[smallest], arr[indx]
            self.min_heapify(smallest, arr, size)

    def insert(self, value):
        """
        Inserts an element in the min heap
        :param value: value to be inserted in the heap
        """
        self.heap.append(value)
        self.heap_size += 1

        indx = self.heap_size - 1

        # Get parent index of the current node
        parent = int(ceil(indx / 2 - 1))

        # Check if the parent value is smaller than the newly inserted value
        # if so, then replace the value with the parent value and check with the new\
parent
        while parent >= 0 and self.heap[indx] < self.heap[parent]:
            self.heap[indx], self.heap[parent] = self.heap[parent], self.heap[indx]
            indx = parent
            parent = int(ceil(indx / 2 - 1))

    def delete(self, indx):
        """
        Deletes the value on the specified index node
        :param indx: index whose node is to be removed
```

```
        :return: Value of the node deleted from the heap
        """
        if self.heap_size == 0:
            print("Heap Underflow!!")
            return

        self.heap[-1], self.heap[indx] = self.heap[indx], self.heap[-1]
        self.heap_size -= 1

        self.min_heapify(indx, self.heap, self.heap_size)

        return self.heap.pop()

    def extract_min(self):
        """
        Extracts the minimum value from the heap
        :return: extracted min value
        """
        return self.delete(0)

    def min(self):
        return self.heap[0]


"""
Algorithm:
*) Split the array stream in 2 halves, min heap(upper array) and max heap (lower arr\
ay)
*) This way the min and max of the heaps will help you get the median fast.
*) Note that the elements should be inserted in the heaps in such a way that the ele\
ments
in lowerMaxHeap are all smaller than all the elements in the upperMinHeap
(i.e., as if the arrays were sorted and then split into two heaps)
"""
n = int(input())
upperMinHeap = MinHeap()
lowerMaxHeap = MaxHeap()

for a_i in range(1, n+1):
    a_t = int(input())

    if lowerMaxHeap.heap_size == 0:  # this case occurs only initially when both hea\
ps are empty
```

```
            lowerMaxHeap.insert(a_t)
    else:
        # Take example stream 1,2,3,4,9,8,7,6,5 to understand the logic
        if upperMinHeap.heap_size == lowerMaxHeap.heap_size:
            if a_t > upperMinHeap.min():
                temp = upperMinHeap.extract_min()
                lowerMaxHeap.insert(temp)
                upperMinHeap.insert(a_t)
            else:
                lowerMaxHeap.insert(a_t)
        elif a_t > lowerMaxHeap.max():
            upperMinHeap.insert(a_t)
        else:
            temp = lowerMaxHeap.extract_max()
            upperMinHeap.insert(temp)
            lowerMaxHeap.insert(a_t)

    # print the median directly if odd number of elements
    # otherwise average of sum of min heap and max heap tops
    num = a_i / 2
    if int(num) != num:
        print(float(lowerMaxHeap.max()))
    else:
        print((lowerMaxHeap.max() + upperMinHeap.min()) / 2)
```

Chapter 5: Linked List

Clone a linked list with next and random pointer

```python
# Python program to Clone a linked list with next and random pointer


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.random = None

class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self, head=None):
        self.head = None

    # Function to create copy the linked list
    def copy_list(self, l2):
        # if first list is empty return
        if self.head is None:
            return

        curr1 = self.head

        # insert new nodes with same data as linst at adjacent positions of each node
        while curr1 is not None:
            node = Node(curr1.data)
            temp = curr1.next
            curr1.next = node
            node.next = temp
            curr1 = temp

        curr1 = self.head
```

```
        # update the random pointers of even node to point to the random nodes of od\
d nodes next
        while curr1 is not None:
            if curr1.random is not None:
                curr1.next.random = curr1.random.next
            curr1 = curr1.next.next

        curr1 = self.head
        curr2 = l2.head
        # assign even nodes to the Copy linked list to make the new list
        # re-assign the old links of list 1
        while curr1 is not None:
            if l2.head is None:
                l2.head = curr1.next
                curr2 = l2.head
            else:
                curr2.next = curr1.next
                curr2 = curr2.next
            curr1.next = curr1.next.next
            curr2.next = None
            curr1 = curr1.next

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        # print data of each node
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next

        current = self.head
        print('None')
        while current is not None:
            print("V", '   ', end='')
            current = current.next

        current = self.head
```

```
        print()
        # print random pointer node's data of each node
        while current is not None:
            if current.random is not None:
                print(current.random.data, '    ', end='')
            else:
                print("N", '    ', end='')
            current = current.next
        print()

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    # make nodes
    node1 = Node(1)
    node2 = Node(2)
    node3 = Node(3)
    node4 = Node(4)
    node5 = Node(5)
    # set next pointer of each node
    node1.next = node2
    node2.next = node3
    node3.next = node4
    node4.next = node5
    # set random pointer of each node
    node1.random = node3
    node2.random = node1
    node3.random = node5
    node4.random = node3
    node5.random = node2
    # assing the nodes list to head
    linked_list.head = node1

    # print the data of linked list
    print('original list:')
    linked_list.print_data()

    # make empty LL2 to copy data of LL1 in
    linked_list2 = SinglyLinkedList(Node())

    # copy LL1 into LL2
    linked_list.copy_list(linked_list2)

    # print the copied linked list
```

```
    print('copied list:')
    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)

```
# Node class
class Node:

    # Constructor to initialise (x, y) coordinates and next
    def __init__(self, x=None, y=None):
        self.x = x
        self.y = y
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to find middle node of a linked list
    def delete_middle_nodes(self):
        current = self.head

        # iterate while the next of the next node is not none
        while current and current.next and current.next.next is not None:
```

```
            # assign variables for next and next of next nodes
            next = current.next
            next_next = current.next.next

            # if x coordinates are equal of current and next node i.e. horizontal li\
ne
            if current.x == next.x:
                # check if there are more than 2 nodes in the horizontal line
                # if yes then delete the middle node and update next and next_next
                while next_next is not None and next.x == next_next.x:
                    current.next = next_next
                    next = next_next
                    next_next = next_next.next
            # if y coordinates are equal of current and next node i.e. vertical line
            elif current.y == next.y:
                # check if there are more than 2 nodes in the vertical line
                # if yes then delete the middle node and update next and next_next
                while next_next is not None and next.y == next_next.y:
                    current.next = next_next
                    next = next_next
                    next_next = next_next.next
            # updated the current node to next node for checking the next line nodes
            current = current.next

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, x, y):
        node = Node(x, y)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print('(',current.x, ',', current.y, ') -> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(40,5)
    linked_list.insert_at_beg(20,5)
    linked_list.insert_at_beg(7,5)
```

```
    linked_list.insert_at_beg(7,10)
    linked_list.insert_at_beg(5,10)
    linked_list.insert_at_beg(1,10)
    linked_list.insert_at_beg(0,10)

    # print the linked list representing vertical and horizontal lines
    linked_list.print_data()

    # call the delete_middle_nodes function
    linked_list.delete_middle_nodes()

    # print the new linked list
    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.

```
# Python program to Construct a Maximum Sum Linked List out of two Sorted Linked Lis\
ts having some Common nodes.

# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to find max sum linked list using 2 linked lists
    def max_sum_list(self, l2):
```

```
        pre1 = cur1 = self.head
        pre2 = cur2 = l2.head
        result = None

        # while any of the lists are not empty keep traversing
        while cur1 or cur2 is not None:
            sum1 = sum2 = 0
            while cur1 and cur2 is not None and cur1.data != cur2.data:

                # if 1st list's node data is less than 2nd list's node
                if cur1.data < cur2.data:
                    sum1 += cur1.data
                    cur1 = cur1.next

                # if 2nd list's node data is less than 1st list's node
                else:
                    sum2 += cur2.data
                    cur2 = cur2.next

            # if any of the list has ended calculate the sum of the other list till \
the end
            if cur1 is None:
                while cur2 is not None:
                    sum2 += cur2.data
                    cur2 = cur2.next
            elif cur2 is None:
                while cur1 is not None:
                    sum1 += cur1.data
                    cur1 = cur1.next

            # initial case when result's head needs to be set
            if pre1 is self.head and pre2 is l2.head:
                result = pre1 if sum1 > sum2 else pre2
            else:
                if sum1 > sum2:
                    pre2.next = pre1.next
                else:
                    pre1.next = pre2.next

            pre1 = cur1
            pre2 = cur2

            if cur1 is not None:
```

```python
                cur1 = cur1.next
            if cur2 is not None:
                cur2 = cur2.next

        return result

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list1 = SinglyLinkedList()
    linked_list1.insert_at_beg(130)
    linked_list1.insert_at_beg(120)
    linked_list1.insert_at_beg(100)
    linked_list1.insert_at_beg(90)
    linked_list1.insert_at_beg(32)
    linked_list1.insert_at_beg(12)
    linked_list1.insert_at_beg(3)
    linked_list1.insert_at_beg(0)

    linked_list2 = SinglyLinkedList()
    linked_list2.insert_at_beg(120)
    linked_list2.insert_at_beg(110)
    linked_list2.insert_at_beg(90)
    linked_list2.insert_at_beg(30)
    linked_list2.insert_at_beg(3)
    linked_list2.insert_at_beg(1)

    print('List 1:')
    linked_list1.print_data()
    print('List 2:')
    linked_list2.print_data()
```

```

    # call the max_sum_list function and update the list 1's head to point the max s\
um list
    linked_list1.head = linked_list1.max_sum_list(linked_list2)

    # print the max sum linked list
    print('Max sum linked list:')
    linked_list1.print_data()
```

Merge a linked list into another linked list at alternate positions

```
# Python program to Merge a linked list into another linked list at alternate positi\
ons


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to merge 2 linked lists at alternate positions
    def merge(self, l2):
        h1 = self.head
        h2 = l2.head

        # Merge at alternate positions until the h1 has alternate positions
        while h1 and h2 is not None:
            h1_next = h1.next
            h2_next = h2.next

```

```
                h1.next = h2
                h2.next = h1_next

                h1 = h1_next
                h2 = h2_next

            # update the head of h2 if linked list remains i.e. no more alternate positi\
ons in h1
            l2.head = h2

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list1 = SinglyLinkedList()
    linked_list1.insert_at_beg(9)
    linked_list1.insert_at_beg(8)
    linked_list1.insert_at_beg(7)
    linked_list1.insert_at_beg(6)
    linked_list1.insert_at_beg(5)

    linked_list2 = SinglyLinkedList()
    linked_list2.insert_at_beg(12)
    linked_list2.insert_at_beg(11)
    linked_list2.insert_at_beg(10)
    linked_list2.insert_at_beg(4)
    linked_list2.insert_at_beg(3)
    linked_list2.insert_at_beg(2)
    linked_list2.insert_at_beg(1)

    print('List 1:')
    linked_list1.print_data()
```

```
    print('List 2:')
    linked_list2.print_data()

    # call the merge function
    linked_list1.merge(linked_list2)

    # print the merged linked list
    print('Merged list:')
    linked_list1.print_data()
    linked_list2.print_data()
```

Perform Merge Sort

```
# Python program to perform Merge Sort on a Signly linked list


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self, head=None):
        self.head = head

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
```

```
            current = current.next
        print('None')


# Function to split the linked list into two halves
def split(head):
    slow = head

    if slow is None or slow.next is None:
        return head, None

    fast = slow.next

    # Reach to the middle of the linked list
    while fast is not None:
        fast = fast.next
        if fast is not None:
            fast = fast.next
            slow = slow.next

    fast = slow.next
    # break the linked list in half
    slow.next = None

    # return the 2 linked lists formed
    return head, fast


# Function to merge linked lists in sorted order
def merge(a, b):
    # Make a dummy node
    dummy = Node()
    # dummy node next will be the head of our merged list
    dummy.next = None

    temp = SinglyLinkedList(dummy)
    tail = temp.head

    while True:
        if a is None:
            tail.next = b
            break
        elif b is None:
```

```
            tail.next = a
            break
        elif a.data <= b.data:
            tail.next = a
            a = a.next
        else:
            tail.next = b
            b = b.next
        tail = tail.next

    return temp.head.next


# Function Merge Sort
def merge_sort(head):

    if head is None or head.next is None:
        return head

    a, b = split(head)
    a = merge_sort(a)
    b = merge_sort(b)

    head = merge(a, b)

    return head


if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)

    # before sorting
    print('before sorting')
    linked_list.print_data()
```

```

    # call merge_sort function
    linked_list.head = merge_sort(linked_list.head)

    # after sorting
    print('after sorting')
    linked_list.print_data()
```

Find Middle Node

```
# Python program to find middle node of a singly linked list


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to find middle node of a linked list
    def find_mid(self):
        fast = self.head
        slow = self.head

        # Make fast run twice the speed of slow
        # when fast is at the last of the list
        # slow will be at the mid node
        while fast is not None:
            fast = fast.next
            if fast is not None:
                fast = fast.next
                slow = slow.next

        return slow
```

```

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    linked_list.print_data()
    # call the find_mid function
    mid = linked_list.find_mid()
    # print the middle node if not None
    if mid is not None:
        print(mid.data)
```

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

```python
# Python program to Point to next higher value node in a linked list with an arbitra\
ry pointer


# Node class
class Node:

    # Constructor to initialise data and next and arbitrary pointer
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.arbit = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self, head=None):
        self.head = head

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        node.arbit = node.next
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        # print data of each node
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next

        current = self.head
        print('None')
        while current is not None:
            print("V", '    ', end='')
            current = current.next

        current = self.head
        print()
```

```python
        # print arbitrary pointer node's data of each node
        while current is not None:
            if current.arbit is not None:
                print(current.arbit.data, '    ', end='')
            else:
                print("N", '    ', end='')
            current = current.next
        print()


# Function to split the linked list into two halves
def split(head):
    slow = head

    if slow is None or slow.arbit is None:
        return head, None

    fast = slow.arbit

    # Reach to the middle of the linked list
    while fast is not None:
        fast = fast.arbit
        if fast is not None:
            fast = fast.arbit
            slow = slow.arbit

    fast = slow.arbit
    # break the linked list in half
    slow.arbit = None

    # return the 2 linked lists formed
    return head, fast


# Function to merge linked lists in sorted order
def merge(a, b):
    # Make a dummy node
    dummy = Node()
    # dummy node arbit will be the head of our merged list
    dummy.arbit = None

    temp = SinglyLinkedList(dummy)
    tail = temp.head
```

```python

    while True:
        if a is None:
            tail.arbit = b
            break
        elif b is None:
            tail.arbit = a
            break
        elif a.data <= b.data:
            tail.arbit = a
            a = a.arbit
        else:
            tail.arbit = b
            b = b.arbit
        tail = tail.arbit

    return temp.head.arbit


# Function Merge Sort
def merge_sort(head):

    if head is None or head.arbit is None:
        return head

    a, b = split(head)
    a = merge_sort(a)
    b = merge_sort(b)

    head = merge(a, b)

    return head


if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(10)
    linked_list.insert_at_beg(5)

    # before linking the arbit
    print('before linking')
```

```python
    linked_list.print_data()

    # call merge_sort function
    # to sort the linked list on the basis of the arbitrary pointers
    merge_sort(linked_list.head)

    # after linking the arbit
    print('after linking')
    linked_list.print_data()
```

Find if linked list contains any cycle

```python
# Python program to check if the singly linked list contains cycle or not


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to find cycle in linked list
    def find_cycle(self):
        fast = self.head.next
        slow = self.head

        # Make fast run twice the speed of slow
        # if fast coincide with slow
        # then there is a loop or cycle
        while fast is not None:
            # return True if cycle
            if fast is slow:
                return True
```

```
            fast = fast.next
            if fast is not None:
                fast = fast.next
                slow = slow.next

        # return False if no cycle
        return False

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)

    temp = head = linked_list.head

    # get pointer to the end of the list
    while temp.next is not None:
        temp = temp.next

    # Make a loop in the list
    temp.next = head.next.next.next

```

```
    # call the find_cycle function
    result = linked_list.find_cycle()

    # print if cycle or not
    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

```
# to select a random node in a singly linked list
import random


# Node class
class Node:
    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to get random node in linked list
    def get_random_node(self):
        if self.head is None:
            return None

        random.seed()
        random_node = self.head
        current = self.head.next
        n = 2
        while current is not None:
            if random.randrange(n) == 0:
                random_node = current
            current = current.next
            n += 1
        return random_node

```

```python
    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)

    random_node = linked_list.get_random_node()
    print("Random node data is:")
    print(random_node.data)
```

Find and remove cycle if any

```python
# Python program to check if the singly linked list contains cycle or not


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to find cycle in linked list
    def find_cycle_remove(self):
        fast = self.head.next
        slow = self.head

        # Make fast run twice the speed of slow
        # if fast coincide with slow
        # then there is a loop or cycle
        while fast is not None:
            # break loop if cycle exists
            if fast is slow:
                break
            fast = fast.next
            if fast is not None:
                fast = fast.next
                slow = slow.next

        if fast is slow:
            slow = self.head
            while fast.next is not slow:
                fast = fast.next
                slow = slow.next
            fast.next = None
            return True

```

```python
        # return False if no cycle
        return False

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)

    temp = head = linked_list.head

    # get pointer to the end of the list
    while temp.next is not None:
        temp = temp.next

    # Make a loop in the list
    temp.next = head.next.next.next

    # call the find_cycle function
    result = linked_list.find_cycle_remove()

    # print if cycle or not
    print('Yes! there was a cycle') if result else print('No! there was no cycle')
```

```
    linked_list.print_data()
```

Reverse sub list of K nodes each

```
# Python program to reverse a linked list in group of given size k


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to reverse K nodes of linked list
    def reverse_k_nodes(self, head, k):
        current = head
        next = None
        prev = None
        count = 0

        # traverse k nodes ahead reversing the links or until current is not None
        while current is not None and count < k:
            next = current.next
            current.next = prev
            prev = current
            current = next
            count += 1

        # recursive call to the function to reverse the remaining n-k nodes
        if next is not None:
            head.next = self.reverse_k_nodes(next, k)

        # return the new header of the current sublist
        return prev
```

```

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    linked_list.print_data()
    # call the reverse k nodes function
    linked_list.head = linked_list.reverse_k_nodes(linked_list.head, 3)
    # print the reversed list
    linked_list.print_data()
```

Reverse alternate sub list of K nodes each

```python
# Python program to alternately reverse a linked list in group of given size k


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to reverse K nodes of linked list
    def reverse_k_nodes(self, head, k):
        current = head
        next = None
        prev = None
        count = 0

        # traverse k nodes ahead reversing the links or until current is not None
        while current is not None and count < k:
            next = current.next
            current.next = prev
            prev = current
            current = next
            count += 1

        if head is not None:
            head.next = current

        count = 0
        # traverse the k nodes to be skipped
        while current is not None and count < k-1:
            current = current.next
            count += 1

        # recursive call to the function to alternate reverse the remaining n-2k nod\
```

```
es
        if current is not None:
            current.next = self.reverse_k_nodes(current.next, k)

        # return the new header of the current sublist
        return prev

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(10)
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    linked_list.print_data()
    # call the reverse k nodes function
    linked_list.head = linked_list.reverse_k_nodes(linked_list.head, 3)
    # print the reversed list
    linked_list.print_data()
```

Reverse a linked list

```python
# Python program to reverse a singly linked list


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to reverse a linked list
    def reverse(self):

        # If linked list is empty
        if self.head is None:
            return None

        current = self.head
        prev = None

        while current is not None:
            # Store the value of current.next
            next = current.next
            # Set current.next to point to the previous node
            current.next = prev
            # Update pointers for next iteration
            prev = current
            current = next

        self.head = prev

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
```

```
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    linked_list.print_data()
    # call the reverse function
    linked_list.reverse()
    # print the reversed list
    linked_list.print_data()
```

Bring even valued nodes to the front

```
# Python program to segregate Even and Odd value nodes in a singly linked list


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
```

```python
    def __init__(self):
        self.head = None

    # Function to segregate even odd value nodes of linked list
    def segregateEvenOdd(self):
        current = None
        prev = self.head
        pivot = None

        # If empty list or single element in the list
        if self.head is None or self.head.next is None:
            return self.head

        # if the first node is even
        # initialise pivot as head
        if prev.data % 2 == 0:
            pivot = self.head
            current = prev.next
        # else find the first node in the list that is even
        # make that node the head of the list
        # initialise pivot as head
        else:
            while prev.next is not None:
                if prev.next.data % 2 == 0:
                    pivot = prev.next
                    prev.next = pivot.next
                    pivot.next = self.head
                    self.head = pivot
                    current = prev.next
                    break
                prev = prev.next

        # keep moving the current pointer and prev pointer
        while current is not None:
            # if even value is found at the node
            if current.data % 2 == 0:
                # if the node is adjacent to pivot
                # simply increment the pivot to next node
                # and shift prev and current one step ahead
                if prev is pivot:
                    pivot = current
                    prev = current
                    current = current.next
```

```
                # else insert the node after the pivot
                # shift the pivot to the newly inserted node
                # update current and prev
                else:
                    prev.next = current.next
                    current.next = pivot.next
                    pivot.next = current
                    pivot = current
                    current = prev.next
            # if odd value simply increment current and prev
            else:
                prev = current
                current = current.next

        # return the updated linked list head
        return self.head

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    print('Before segregation:', end=' ')
    linked_list.print_data()
    linked_list.head = linked_list.segregateEvenOdd()
```

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

Implementation of Singly Linked List

```
# class of a node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


# class of the signly linked list
class SinglyLinkedList:
    def __init__(self):
        self.head = None

    # function to calculate size of the list
    def size(self):
        # initialize temporary variable and size to zero
        current = self.head
        size = 0

        # count until current does not reach the end of the list i.e. NULL or None
        while current is not None:
            size += 1
            current = current.next
        return size

    # function to insert at the end of the list
    def insert_at_end(self, data):
        # Create the new node
        node = Node(data)

        # If the list is empty
        if self.head is None:
            self.head = node
        else:
            current = self.head

            # Otherwise move to the last node of the list
            while current.next is not None:
```

```
                current = current.next

            # Point the last node of the list to the new node
            # So that the new node gets added to the end of the list
            current.next = node

    # function to insert at the beginning of the list
    def insert_at_beg(self, data):
        node = Node(data)
        # Next pointer of the new node points to the head
        node.next = self.head

        # Updated the head as new node
        self.head = node

    # function to delete from the end of the list
    def delete_from_end(self):
        current = self.head
        previous = None

        if current is None:
            print("Linked List Underflow!!")
        else:
            while current.next is not None:
                previous = current
                current = current.next

            if previous is None:
                self.head = None
            else:
                previous.next = None

    # function to delete from the beginning of the list
    def delete_from_beg(self):
        current = self.head

        if current is None:
            print("Linked List Underflow!!")
        else:
            self.head = current.next

    # function to print the linked list data
    def print_data(self):
```

```python
        current = self.head

        while current is not None:
            print(current.data, '->', end='')
            current = current.next
        print('End of list')


# Main program:
if __name__ == '__main__':
    # Create a singly linked list object
    linked_list = SinglyLinkedList()

    # Insert at the beginning 3, 2, 1
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)
    print('After insertion at the beginning:')
    linked_list.print_data()

    # Insert at the end of the list 4, 5
    linked_list.insert_at_end(4)
    linked_list.insert_at_end(5)
    print('After insertion at the end:')
    linked_list.print_data()

    # Delete 1 from the beginning
    print('After deletion at the beginning:')
    linked_list.delete_from_beg()
    linked_list.print_data()

    # Delete 5 from the end
    print('After deletion at the end:')
    linked_list.delete_from_end()
    linked_list.print_data()

    # print size of the list
    print("size: ", linked_list.size())
```

Skip M nodes and then delete N nodes alternately

```
# Python program to skip M nodes and then delete N nodes alternately in a singly lin\
ked list


# Node class
class Node:

    # Constructor to initialise data and next
    def __init__(self, data=None):
        self.data = data
        self.next = None


class SinglyLinkedList:

    # Constructor to initialise head
    def __init__(self):
        self.head = None

    # Function to skip M delete N nodes
    def skip_m_delete_n(self, m, n):
        current = self.head

        # if list is empty return
        if current is None:
            return

        # Main loop to traverse the whole list
        while current is not None:
            # loop to skip M nodes
            for i in range(1, m):
                if current is None:
                    return
                current = current.next

            if current is None:
                return

            # loop to delete N nodes
            temp = current.next
            for i in range(1, n+1):
                if temp is None:
                    break
```

```
                temp = temp.next

            # Point the last node skipped to the node after N nodes deletion
            current.next = temp
            # set current for next iteration
            current = temp

    # Function to Insert data at the beginning of the linked list
    def insert_at_beg(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # Function to print the linked list
    def print_data(self):
        current = self.head
        while current is not None:
            print(current.data, '-> ', end='')
            current = current.next
        print('None')

if __name__ == '__main__':
    linked_list = SinglyLinkedList()
    linked_list.insert_at_beg(9)
    linked_list.insert_at_beg(8)
    linked_list.insert_at_beg(7)
    linked_list.insert_at_beg(6)
    linked_list.insert_at_beg(5)
    linked_list.insert_at_beg(4)
    linked_list.insert_at_beg(3)
    linked_list.insert_at_beg(2)
    linked_list.insert_at_beg(1)

    linked_list.print_data()

    # call the skip_m_delete_n function
    linked_list.skip_m_delete_n(2, 2)

    # print the modified linked list
    linked_list.print_data()
```

Chapter 6: Mathematics

Fine the number of trailing zeros in factorial of a number

```python
# Find the number of trailing zeros present in the factorial of a number n.

def fact_trailing_zeros(num):
    c = 5
    count = 0

    # count number of factors of 5 possible in num!
    # 5 paired with 2 will give 10 i.e. 1 trailing zero
    # powers of 5 will give multiple zeros
    while num // c != 0:
        count += num // c
        c *= 5

    return count

num = 1000
print(f"{num}! has {fact_trailing_zeros(num)} trailing zeros")
```

Find the greatest common divisor of 2 numbers

```python
# Basic eucledian algorithm to find the greatest common divisor of 2 numbers



def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)

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".

```python
# print all prime factors of a given number


from math import sqrt


def prime_factors(num):
    # list to store the prime factors
    prime_factor_lis = []

    # if 2 is a factor of the number
    while num % 2 == 0:
        prime_factor_lis.append(2)
        num /= 2

    for i in range(3, int(sqrt(num)), 2):
        while num % i == 0:
            prime_factor_lis.append(i)
            num /= i

    return prime_factor_lis

if __name__ == '__main__':
    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.

```
# Given a number n, print all primes smaller than or equal to n. It is also given th\
at n is a small number.

# function to find prime numbers less than or equal to num
def find_primes_sieve(num):
    # list of all numbers upto n
    intList = [True for i in range(num+1)]

    # first prime
    p = 2

    while p * p <= num:

        # if intList[p] is True means its a prime number
        if intList[p]:
            for i in range(p**2, num+1, p):
                intList[i] = False

        p += 1

    lis = []
    for i in range(2, len(intList)):
        if intList[i]:
            lis.append(i)

    return lis

if __name__ == '__main__':
    primes = find_primes_sieve(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.

```
def check_threat(king_x, king_y, queen_x, queen_y):
    # If coordinates are non-integer or outside the bounds of the chessboard
    if not (validate(king_x) and validate(king_y) and validate(queen_x) and validate\
(queen_y)):
        return False

    # if king is in the vertical column of queen, king_x = queen_x
    # if king is in the horizontal row of queen, king_y = queen_y
    # if king is in the diagonal of queen, abs(king_y - queen_y) = abs(king_x - quee\
n_x) because the will form a square
    if king_x == queen_x or king_y == queen_y or abs(king_y - queen_y) == abs(king_x\
 - queen_x):
        return True

    return False


def validate(coordinate):
    if type(coordinate) is int and 1 <= coordinate <= 8:
        return True

    return False


print("Queen threatens King") if check_threat(1, 1, 5, 5) else print("Queen does not\
 threaten King")
```

Search in a row wise and column wise sorted matrix

Given an n x 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.

```python
def search(mat, x):
    n = len(mat)
    i = 0
    j = n-1
    while i < n and j >= 0:
        if mat[i][j] == x:
            return i, j
        elif mat[i][j] < x:
            i += 1
        else:
            j -= 1
    return [-1]


matrix = [[10, 20, 30, 40],
          [15, 25, 35, 45],
          [27, 29, 37, 48],
          [32, 33, 39, 50]
        ]

print("Enter element you want to search: ")
element = int(input())
index = search(matrix, element)
if len(index) == 1:
    print("Element not found")
else:
    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

```python
# Given a 2D array, print it in spiral form.

def print_spiral(mat):
    row, col = len(mat), len(mat[0])
    k = 0
    l = 0
    while k < row and l < col:
        for i in range(col):
            print(mat[k][i], end=' ')
        k += 1

        for i in range(k, row):
            print(mat[i][col-1], end=' ')
        col -= 1

        if k < row:
            for i in reversed(range(l, col-1)):
                print(mat[row-1][i], end=' ')
            row -= 1

        if l < col:
            for i in reversed(range(k, row-1)):
                print(mat[i][l], end=' ')
            l += 1


a = [[1, 2, 3, 4, 5, 6],
     [7, 8, 9, 10, 11, 12],
     [13, 14, 15, 16, 17, 18]
     ]

print_spiral(a)
```

Chapter 8: Strings or Arrays

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

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


def longest_k_unique(string, k):
    unique = 0
    sets = set({})

    for i in string:
        if i not in sets:
            sets.add(i)
            unique += 1

    if unique < k:
        return -1, -1

    count = [0] * 26
    curr_end = curr_start = max_window_start = 0
    max_window_len = 1

    count[ord(string[0]) - ord('a')] += 1
    for i in range(1, len(string)):
        count[ord(string[i]) - ord('a')] += 1
        curr_end += 1

        while not isValid(count, k):
            count[ord(string[curr_start]) - ord('a')] -= 1
            curr_start += 1

        if curr_end - curr_start + 1 > max_window_len:
            max_window_len = curr_end - curr_start + 1
            max_window_start = curr_start

    return max_window_start, max_window_len
```

```


def isValid(count, k):
    val = 0
    for i in count:
        if i > 0:
            val += 1

    return k >= val


if __name__ == '__main__':
    string = 'aabaabab'
    k = 3
    max_start, max_len = longest_k_unique(string, k)

    if max_len == -1:
        print("K unique characters sub string does not exist.")
    else:
        print('max string with {} unique characters is "'.format(k) + string[max_sta\
rt: max_start + max_len] +
              '" of length', max_len)
```

Find a pattern in a string using KMP search algorithm

```
# To find indexes of String "Pattern" in a given String "Text" using KMP Algorithm


# function that returns list of indexes where the patters matches
def KMP_Search(pattern, text):
    n = len(text)
    m = len(pattern)

    # pre-compute prefix array of the pattern
    prefix_arr = get_prefix_arr(pattern, m)

    # stores start point of pattern match in text
    start_points = []

    i = 0
    j = 0

```

```
    # while the whole text has not been searched
    while i != n:
        # if the character in text matches the pattern character
        if text[i] == pattern[j]:
            i += 1
            j += 1
        # else find the previous index from where the matching can resume
        else:
            j = prefix_arr[j-1]

        # if pattern length has been reached that means a pattern has been found
        if j == m:
            start_points.append(i-j)
            j = prefix_arr[j-1]
        elif j == 0:
            i += 1
    # return the starting position of pattern in text
    return start_points


# pre-computes the prefix array for KMP search
def get_prefix_arr(pattern, m):
    prefix_arr = [0] * m
    j = 0
    i = 1

    while i != m:
        if pattern[i] == pattern[j]:
            j += 1
            prefix_arr[i] = j
            i += 1
        elif j != 0:
                j = prefix_arr[j-1]
        else:
            prefix_arr[i] = 0
            i += 1

    return prefix_arr

txt = "ABABDABACDABABCABABCABAB"
pat = "ABABCABAB"

start_indexes = KMP_Search(pat, txt)
```

```python

for i in start_indexes:
    print(i)
```

Find the Kth smallest element in the array

Approach used: QuickSelect
Time Complexity: O(n)

```python
# Find the Kth smallest element in a given array.
# taking smallest in arr as 1st smallest

def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]  # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:
            # increment index of smaller element
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quick_select(arr, low, high, k):
    # arr follows zero indexing hence kth smallest will be at index (k - 1)
    k -= 1
    while low < high:
        p_index = partition(arr, low, high)

        # found the kth smallest value
        if p_index == k:
            return arr[p_index]
        # pivot index is less than k hence kth smallest is in the right half
        elif p_index < k:
            low = p_index + 1
        # pivot index is greater than k hence kth smallest is in the left half
```

```
        else:
            high = p_index - 1
    # if k < 0 or k > len(arr) then simply return the smallest or largest value in a\
rr
    return arr[low]


arr = [10, 7, 8, 9, 1, 5]
n = len(arr) - 1

# find 4th smallest element in the array
print(quick_select(arr, 0, n, 4))
```

Find a pair in an array with sum x

```
# Find a pair of elements in the array with sum = x

"""
Method 1: If unsorted array
Time Complexity: O(n)
Space Complexity: O(n)
"""


def find_pair_unsorted(arr, x):
    elem_set = set({})

    # To store the indexes of both the elements
    pair = [-1, -1]

    for value in arr:
        # if x - value has already been discovered in the array
        # Pair found, return the values
        if (x-value) in elem_set:
            return x-value, value

        # else add the current value in the elem_set
        else:
            elem_set.add(value)

    return "Not found"

```

```
arr = [1, 4, 45, 6, 10, 8]
print('Unsorted array:', arr)
print('Pair with sum 16 in unsorted array:', find_pair_unsorted(arr, 16))


"""
Method 2: If array is sorted
Time Complexity: O(n)
Space Complexity: O(1)
"""


def find_pair_sorted(arr, x):
    # initialize variables to the start and end of the array
    l = 0
    r = len(arr) - 1

    while l < r:
        pair_sum = arr[l] + arr[r]

        # if pair is found
        if pair_sum == x:
            return arr[l], arr[r]
        # if the pair sum is less than x go to the next bigger value from left
        elif pair_sum < x:
            l += 1
        # if the pair sum is more than x go to the next lesser value from right
        else:
            r -= 1

    # If pair not found
    return "Not found"


arr = [2, 6, 10, 15, 18, 20, 23, 25]
print('Sorted array:', arr)
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.

```
def addParentheses(str_arr, leftParenCount, rightParenCount, combinations, index):
    if leftParenCount < 0 or rightParenCount < 0:
        return
    if leftParenCount == 0 and rightParenCount == 0:
        combinations.append(''.join(str_arr))
    else:
        if leftParenCount > 0:
            str_arr[index] = '('
            addParentheses(str_arr, leftParenCount - 1, rightParenCount, combination\
s, index + 1)

        if rightParenCount > leftParenCount:
            str_arr[index] = ')'
            addParentheses(str_arr, leftParenCount, rightParenCount - 1, combination\
s, index + 1)


def generateParentheses(count):
    str_arr = [''] * (count * 2)
    combinations = []
    addParentheses(str_arr, count, count, combinations, 0)
    return combinations


parenthesis_pairs = 3
combinations = generateParentheses(parenthesis_pairs)
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’]

```
def reverse_sentence(arr):
    # reverse all characters:
    n = len(arr)
    mirrorReverse(arr, 0, n-1)

    # reverse each word:
    word_start = None
    for i in range(0, n):
        if arr[i] == " ":
            if word_start is not None:
                mirrorReverse(arr, word_start, i-1)
                word_start = None
        elif i == n-1:
            if word_start is not None:
                mirrorReverse(arr, word_start, i)
        else:
            if word_start is None:
                word_start = i


def mirrorReverse(arr, start, end):
    while start < end:
        tmp = arr[start]
        arr[start] = arr[end]
        arr[end] = tmp
        start += 1
        end -= 1


if __name__ == '__main__':
    arr = ['p', 'e', 'r', 'f', 'e', 'c', 't', ' ', 'm', 'a', 'k', 'e', 's', ' ', 'p'\
, 'r', 'a', 'c', 't', 'i', 'c', 'e']

    print('Before reverse:', ''.join(arr))
    reverse_sentence(arr)
    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?

```python
def binarySearch(arr, num, begin, end):
    while begin <= end:
        mid = round((begin+end)/2)
        if arr[mid] < num:
            begin = mid + 1
        elif arr[mid] == num:
            return mid
        else:
            end = mid - 1
    return -1


def shiftedArrSearch(shiftArr, num):
    originalFirst = getOrigFirst(shiftArr)
    n = len(shiftArr)
    if shiftArr[originalFirst] <= num <= shiftArr[n-1]:
        return binarySearch(shiftArr, num, originalFirst, n - 1)
    else:
        return binarySearch(shiftArr, num, 0, originalFirst - 1)


def getOrigFirst(arr):
    begin = 0
    end = len(arr)-1
    while begin <= end:
        mid = int((begin+end)/2)
        if mid == 0 or arr[mid] < arr[mid-1]:
            return mid
        if arr[mid] > arr[0]:
            begin = mid + 1
        else:
            end = mid - 1
    return 0

if __name__ == '__main__':
    shiftArr = [9, 12, 17, 2, 4, 5]
    print(shiftedArrSearch(shiftArr, 4))
```

Print all permutations of a given string

```python
# To print all permutations of a given string

count = 0


def permutations(mat, l, r):
    if l == r:
        print(''.join(mat))
        global count
        count += 1
    else:
        for i in range(l, r+1):
            mat[l], mat[i] = mat[i], mat[l]
            permutations(mat, l+1, r)
            mat[l], mat[i] = mat[i], mat[l]


string = "ABC"
permutations(list(string), 0, len(string)-1)
print('total permutations:', count)
```

Linear Search in an array

```python
# Function for linear search
# inputs: array of elements 'arr', key to be searched 'x'
# returns: index of first occurrence of x in arr
def linear_search(arr, x):
    # traverse the array
    for i in range(0, len(arr)):

        # if element at current index is same as x
        # return the index value
        if arr[i] == x:
            return i

    # if the element is not found in the array return -1
    return -1

arr = [3, 2, 1, 5, 6, 4]
print(linear_search(arr, 1))
```

Binary Search in an array

```
# Function for binary search
# inputs: sorted array 'arr', key to be searched 'x'
# returns: index of the occurrence of x found in arr


def binary_search(arr, x):
    l = 0
    r = len(arr)

    # while the left index marker < right index marker
    while l < r:
        # find the index of the middle element
        mid = int(l + ((r - l) / 2))

        # if middle element is x, return mid
        if arr[mid] == x:
            return mid

        # if middle element is < x, update l to search to the right of mid
        elif arr[mid] < x:
            l = mid + 1

        # if middle element is > x, update r to search to the left of mid
        else:
            r = mid - 1

    return -1


arr = [1, 4, 5, 7, 8, 10, 13, 15]
print(binary_search(arr, 5))
```

Interpolation Search in an array

```python
def interpolation_search(arr, key):
    low = 0
    high = len(arr) - 1

    while arr[high] != arr[low] and key >= arr[low] and key <= arr[high]:
        mid = int(low + ((key - arr[low]) * (high - low) / (arr[high] - arr[low])))

        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1

    return -1


# input arr
arr = [2, 4, 6, 8, 10, 12, 14, 16]

# interpolation_search call to search 3 in arr
print('6 is at index: ', interpolation_search(arr, 6))

# 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]

```python
# bubble sort function
def bubble_sort(arr):
    n = len(arr)

    # Repeat loop N times
    # equivalent to: for(i = 0; i < n-1; i++)
    for i in range(0, n-1):
        # Repeat internal loop for (N-i)th largest element
        for j in range(0, n-i-1):

            # if jth value is greater than (j+1) value
```

```python
            if arr[j] > arr[j+1]:
                # swap the values at j and j+1 index
                # Pythonic way to swap 2 variable values -> x, y = y, x
                arr[j], arr[j+1] = arr[j+1], arr[j]


arr = [64, 34, 25, 12, 22, 11, 90]
print('Before sorting:', arr)

# call bubble sort function on the array
bubble_sort(arr)

print('After sorting:', arr)
```

Counting sort Algorithm (non-comparision based sorting)

```python
# counting sort without stable sorting


def counting_sort(arr):
    n = len(arr)

    # get the maximum value of the array
    max_val = max(arr)

    count = [0] * (max_val + 1)

    # fill the count array
    # for each element x in the array
    for x in arr:
        count[x] += 1

    k = 0
    for i in range(0, len(count)):
        for j in range(0, count[i]):
            arr[k] = i
            k += 1


arr = [3, 2, 1, 3, 2, 5, 5, 3]
```

```python
print('Before counting sort:', arr)

# call counting sort function
counting_sort(arr)

print('After counting sort:', arr)
```

Insertion sort Algorithm

Output:

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

```python
def insertion_sort(arr):
    n = len(arr)

    for i in range(1, n):
        x = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > x:
            # copy value of previous index to index + 1
            arr[j + 1] = arr[j]
            # j = j - 1
            j -= 1
        # copy the value which was at ith index to its correct sorted position
        arr[j + 1] = x


arr = [12, 11, 13, 5, 6]
print('Before sort arr: ', arr)
insertion_sort(arr)
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.

```
import heapq


def kHeapSort(arr, k):
    h = []
    n = len(arr)
    for i in range(0, k+1):
      heapq.heappush(h, arr[i])
    for i in range(k+1, n):
        arr[i-(k+1)] = heapq.heappop(h)
        heapq.heappush(h, arr[i])
    for i in range(0, k+1):
     arr[n-k-1 + i] = heapq.heappop(h)
    return arr

if __name__ == '__main__':
    arr = [2, 1, 4, 3, 6, 5]

    print(kHeapSort(arr, 2))
```

Merge Sort Algorithm

```
# Program to perform merge sort on an array


def merge(arr, low, mid, high):
    n1 = mid - low + 1
    n2 = high - mid

    # create temporary arrays
    """
    arr = [0] * n is equivalent to:
    arr = [0, 0, 0, ..., 0]
    array of n zeros
    """
    arr1 = [0] * n1
    arr2 = [0] * n2

    # copy data of arr into arr1 and arr2
    for i in range(0, n1):
        arr1[i] = arr[low + i]

```

```python
    for i in range(0, n2):
        arr2[i] = arr[mid + 1 + i]

    # initialize i, j to 0
    i = j = 0

    # initialize k to lower index
    k = low

    # merge the 2 arrays
    while i < n1 and j < n2:
        if arr1[i] < arr2[j]:
            arr[k] = arr1[i]
            i += 1
        else:
            arr[k] = arr2[j]
            j += 1
        k += 1

    # if elements left in arr1 copy them to arr
    while i < n1:
        arr[k] = arr1[i]
        i += 1
        k += 1

    # if elements left in arr2 copy them to arr
    while j < n2:
        arr[k] = arr2[j]
        j += 1
        k += 1


def merge_sort(arr, low, high):
    if low < high:
        # mid = int((low + high) / 2)
        mid = int(low + ((high - low) / 2))

        # call merge_sort on 2 halves
        merge_sort(arr, low, mid)
        merge_sort(arr, mid+1, high)

        # merge the two sorted halves
        merge(arr, low, mid, high)
```

```python


arr = [5, 21, 7, 3, 4, 8, 9, 10, 100, 15]

print('Before merge sort:', arr)

# Call merge sort on arr
merge_sort(arr, 0, len(arr)-1)

print('After merge sort:', arr)
```

Quick Sort Algorithm using last element as pivot

```python
# Python program for implementation of Quicksort Sort by taking last element as pivot


def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]  # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:
            # increment index of smaller element
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


# Function to do Quick sort
def quickSort(arr, low, high):
    if low < high:
        # pivot is set to its right position after partition call
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi - 1)
```

```
        quickSort(arr, pi + 1, high)


# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
print("Before sorting array is:")
for i in range(n):
    print(arr[i], end=' -> ')
print('end')

quickSort(arr, 0, n - 1)

print("Sorted array is:")
for i in range(n):
    print(arr[i], end=' -> ')
print('end')
```

Selection sort Algorithm

```
# selection sort function
def selection_sort(arr):
    n = len(arr)
    for i in range(0, n):
        for j in range(i+1, n):
            # if the value at i is > value at j -> swap
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]


# input arr
arr = [3, 2, 4, 1, 5]
print('Before selection sort:', arr)

# call selection sort function
selection_sort(arr)
print('After selection sort:', arr)
```

Chapter 9: Tree

Binary Search Tree implementation

```python
#!/usr/bin/python3


class Node:
    def __init__(self, data=None, right_child=None, left_child=None, parent=None):
        self.data = data
        self.right_child = right_child
        self.left_child = left_child
        self.parent = parent

    def has_left_child(self):
        return self.left_child

    def has_right_child(self):
        return self.right_child

    def has_both_children(self):
        return self.right_child and self.left_child

    def has_any_children(self):
        return self.right_child or self.left_child

    def is_root(self):
        return not self.parent

    def is_leaf(self):
        return not(self.right_child or self.left_child)

    def is_left_child(self):
        return self.parent and self.parent.left_child == self

    def is_right_child(self):
        return self.parent and self.parent.right_child == self

    def replace_data(self, data):
```

```
        self.data = data

    def replace_left_child(self, left_child):
        self.left_child = left_child

    def replace_right_child(self, right_child):
        self.right_child = right_child


class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert_node(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(data, self.root)

    def _insert(self, data, current_node):
        if data < current_node.data:
            if current_node.has_left_child():
                self._insert(data, current_node.left_child)
            else:
                current_node.left_child = Node(data, parent=current_node)
        else:
            if current_node.has_right_child():
                self._insert(data, current_node.right_child)
            else:
                current_node.right_child = Node(data, parent=current_node)

    @staticmethod
    def find_inorder_ancestor(current_node):
        if current_node.has_right_child():
            current_node = current_node.right_child
            while current_node.has_left_child():
                current_node = current_node.left_child
            return current_node

        ancestor = current_node.parent
        child = current_node
        while ancestor is not None and child is ancestor.right_child:
            child = ancestor
```

```
            ancestor = child.parent
        return ancestor

    def delete_node(self, data):
        if self.root is None:
            print("No node to delete in the tree")
        else:
            if self._delete(data, self.root) is not None:
                print("Deleted", data)
            else:
                print(data,"not found")

    def _delete(self, data, current_node):
        while current_node is not None and data != current_node.data:
            if data <= current_node.data:
                current_node = current_node.left_child
            else:
                current_node = current_node.right_child

        if current_node is not None:
            if current_node.is_leaf():
                if current_node.is_left_child():
                    current_node.parent.left_child = None
                else:
                    current_node.parent.right_child = None
            else:
                successor = self.find_inorder_ancestor(current_node)
                if successor is None:
                    current_node.data = current_node.left_child.data
                    current_node.left_child = None
                else:
                    temp = current_node.data
                    current_node.data = successor.data
                    successor.data = temp
                    self._delete(temp, successor)
            return True
        else:
            return None

    def inorder(self):
        current_node = self.root
        self._inorder(current_node)
        print('End')
```

```python

    def _inorder(self, current_node):
        if current_node is None:
            return
        self._inorder(current_node.left_child)
        print(current_node.data," -> ",end='')
        self._inorder(current_node.right_child)

if __name__ == '__main__':
    tree = BinarySearchTree()
    tree.insert_node(6)
    tree.insert_node(9)
    tree.insert_node(6)
    tree.insert_node(5)
    tree.insert_node(8)
    tree.insert_node(7)
    tree.insert_node(3)
    tree.insert_node(2)
    tree.insert_node(4)
    tree.inorder()
    tree.delete_node(6)
    tree.delete_node(1)
    tree.delete_node(3)
    tree.inorder()
```

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

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


def check_pre_order(arr):
    minimum = -float('inf')

    stack = []

    for val in arr:
        if val < minimum:
            return False

        while len(stack) != 0 and stack[-1] < val:
```

```
            minimum = stack.pop()

        stack.append(val)

    return True


if __name__ == '__main__':
    pre_order = [40, 30, 35, 80, 100]
    print("Valid Preorder" if check_pre_order(pre_order) else "Invalid Preorder")
    pre_order = [40, 30, 35, 20, 80, 100]
    print("Valid Preorder" if check_pre_order(pre_order) else "Invalid Preorder")
```

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

```
# find the in-order ancestor of a given node in BST


# A Binary Search Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data, parent=None):
        self.data = data
        self.left = None
        self.right = None
        self.parent = parent


def findInOrderAncestor(n):
    if n.right is not None:
        return findMinKeyWithinTree(n.right)

    ancestor = n.parent
    child = n
    while ancestor is not None and child == ancestor.right:
        child = ancestor
        ancestor = child.parent
    return ancestor


def findMinKeyWithinTree(root):
    while root.left is not None:
```

```
        root = root.left
    return root


if __name__ == '__main__':
    root = Node(4)
    root.left = Node(2, root)
    root.right = Node(6, root)
    root.left.left = Node(1, root.left)
    root.left.right = Node(3, root.left)
    successor = findInOrderAncestor(root.left.right)

    if successor is not None:
        print(successor.data)
    else:
        print("No in order successor")
```

Find the Lowest Common Ancestor

```
# Find the Lowest Common Ancestor (LCA) in a Binary Search Tree


# A Binary Search Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    def insert_node(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(data, self.root)

    def _insert(self, data, current_node):
```

```
        if data <= current_node.data:
            if current_node.left is not None:
                self._insert(data, current_node.left)
            else:
                current_node.left = Node(data)
        else:
            if current_node.right is not None:
                self._insert(data, current_node.right)
            else:
                current_node.right = Node(data)

    def inorder(self):
        current_node = self.root
        self._inorder(current_node)
        print('End')

    def _inorder(self, current_node):
        if current_node is None:
            return
        self._inorder(current_node.left)
        print(current_node.data, " -> ", end='')
        self._inorder(current_node.right)


# assuming both nodes are present in the tree
def lca_bst(root, value1, value2):
    while root is not None:
        if value2 > root.data < value1:
            root = root.right
        elif value2 < root.data > value1:
            root = root.left
        else:
            return root.data


if __name__ == '__main__':
    tree = BST()
    tree.insert_node(6)
    tree.insert_node(8)
    tree.insert_node(9)
    tree.insert_node(6)
    tree.insert_node(5)
    tree.insert_node(7)
```

```
    tree.insert_node(3)
    tree.insert_node(2)
    tree.insert_node(4)
    print(lca_bst(tree.root, 4, 2))

"""
given tree:
                6
            6          8
        5       7          9
     3
  2     4
"""
```

Given a sorted array, create a BST with minimal height

```
# Given a sorted array, create a binary search tree with minimal height


# A Binary Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def make_minimal_bst(arr, start, end):
    if end < start:
        return None
    mid = int((start + end) / 2)
    root = Node(arr[mid])

    root.left = make_minimal_bst(arr, start, mid-1)
    root.right = make_minimal_bst(arr, mid+1, end)

    return root


def in_order(root):
    if root is None:
```

```python
        return
    in_order(root.left)
    print(root.data, end=' -> ')
    in_order(root.right)

if __name__ == '__main__':
    arr = [1, 2, 3, 4, 5, 6, 7, 8]
    root = make_minimal_bst(arr, 0, len(arr)-1)
    in_order(root)
    print('end')
```

Print Nodes in Bottom View of Binary Tree

```python
# Print Nodes in Bottom View of Binary Tree
from collections import deque


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def bottom_view(root):
    if root is None:
        return

    # make an empty queue for BFS
    q = deque()

    # dict to store bottom view keys
    bottomview = {}

    # append root in the queue with horizontal distance as 0
    q.append((root, 0))

    while q:
        # get the element and horizontal distance
        elem, hd = q.popleft()

        # update the last seen hd element
```

```
        bottomview[hd] = elem.data

        # add left and right child in the queue with hd - 1 and hd + 1
        if elem.left is not None:
            q.append((elem.left, hd - 1))
        if elem.right is not None:
            q.append((elem.right, hd + 1))

    # return the bottomview
    return bottomview


if __name__ == '__main__':
    root = Node(20)
    root.left = Node(8)
    root.right = Node(22)
    root.left.left = Node(5)
    root.left.right = Node(3)
    root.right.left = Node(4)
    root.right.right = Node(25)
    root.left.right.left = Node(10)
    root.left.right.right = Node(14)

    bottomview = bottom_view(root)

    for i in sorted(bottomview):
        print(bottomview[i], end=' ')
```

Check if a binary tree is height balanced

```
# Check if a binary tree is height balanced
# abs(height[leftTree] - height[rightTree]) <= 1


# A Binary Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

```

```python

def check_height(root):
    if root is None:
        return -1
    left_height = check_height(root.left)
    if left_height is float('inf'):
        return float('inf')

    right_height = check_height(root.right)
    if right_height is float('inf'):
        return float('inf')

    height = abs(left_height - right_height)
    if height > 1:
        return float('inf')
    else:
        return max(left_height, right_height) + 1


def isBalanced(root):
    return check_height(root) != float('inf')

if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    if isBalanced(root):
        print("Yes! the tree is balanced")
    else:
        print("No! the tree is not balanced")
```

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

```python
# Check whether a binary tree is a full binary tree or not


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def check_full_bt(root):
    # If tree is empty
    if root is None:
        return True

    # if both child are None
    if root.left is None and root.right is None:
        return True

    # if the node has both children and both sub tree are full binary trees
    if root.left is not None and root.right is not None:
        return check_full_bt(root.left) and check_full_bt(root.right)

    return False

if __name__ == '__main__':
    root = Node(10)
    root.left = Node(20)
    root.right = Node(30)

    root.left.right = Node(40)
    root.left.left = Node(50)
    root.right.left = Node(60)
    root.right.right = Node(70)

    root.left.left.left = Node(80)
    root.left.left.right = Node(90)
    root.left.right.left = Node(80)
    root.left.right.right = Node(90)
    root.right.left.left = Node(80)
    root.right.left.right = Node(90)
    root.right.right.left = Node(80)
    root.right.right.right = Node(90)
```

```python

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

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

```python
# Given two binary trees, check if the first tree is subtree of the second one.


# A Binary Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def store_in_order(root, arr):
    if root is None:
        arr.append('$')
        return
    store_in_order(root.left, arr)
    arr.append(root.data)
    store_in_order(root.right, arr)


def store_pre_order(root, arr):
    if root is None:
        arr.append('$')
        return
    store_pre_order(root.left, arr)
    store_pre_order(root.right, arr)
    arr.append(root.data)


def isSubTree(t1, t2):
    order_t1 = []
```

```python
    order_t2 = []

    store_in_order(t1, order_t1)
    store_in_order(t2, order_t2)
    if ''.join(order_t1).find(''.join(order_t2)) == -1:
        return False

    order_t1 = []
    order_t2 = []

    store_pre_order(t1, order_t1)
    store_pre_order(t2, order_t2)
    if ''.join(order_t1).find(''.join(order_t2)) == -1:
        return False

    return True


T = Node('a')
T.left = Node('b')
T.right = Node('d')
T.left.left = Node('c')
T.left.right = Node('e')

S = Node('b')
S.left = Node('c')
S.right = Node('e')

if isSubTree(T, S):
    print('Yes, S is a subtree of T')
else:
    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

```python
# A binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# This function returns reference to LCA of two given values n1 and n2
# v1 is set as true by this function if n1 is found
# v2 is set as true by this function if n2 is found
def findLCAUtil(root, n1, n2, v):
    # Base Case
    if root is None:
        return None

    # IF either n1 or n2 matches ith root's key, report
    # the presence by setting v1 or v2 as true and return
    # root
    if root.key == n1:
        v[0] = True
        return root

    if root.key == n2:
        v[1] = True
        return root

    # Look for keys in left and right subtree
    left_lca = findLCAUtil(root.left, n1, n2, v)
    right_lca = findLCAUtil(root.right, n1, n2, v)

    # if one key is present in left subtree and other is present in other,
    # So this node is the LCA
    if left_lca and right_lca:
        return root

    # Otherwise check if left subtree or right subtree is LCA
    return left_lca if left_lca is not None else right_lca


def find(root, k):
    # Base Case
```

```
    if root is None:
        return False

    # If key is present at root, or if left subtree or right
    # subtree , return true
    if (root.key == k or find(root.left, k) or
            find(root.right, k)):
        return True

    # Else return false
    return False


# This function returns LCA of n1 and n2 only if both
# n1 and n2 are present in tree, otherwise returns None
def findLCA(root, n1, n2):
    # Initialize n1 and n2 as not visited
    v = [False, False]

    # Find lca of n1 and n2
    lca = findLCAUtil(root, n1, n2, v)

    # Returns LCA only if both n1 and n2 are present in tree
    if v[0] and v[1] or v[0] and find(lca, n2) or v[1] and find(lca, n1):
        return lca

    # Else return None
    return None


# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

lca = findLCA(root, 4, 5)

if lca is not None:
    print("LCA(4, 5) = ", lca.key)
```

```python
else:
    print("Keys are not present")

lca = findLCA(root, 4, 10)
if lca is not None:
    print("LCA(4,10) = ", lca.key)
else:
    print("Keys are not present")
```

Create a list of all nodes at each depth

```python
# Create a list of all nodes at each depth

# A Binary Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def list_of_depths(root):
    if root is None:
        return []
    depths = []
    q = []
    q.append(root)

    while q:
        parents = q
        depths.append(q)
        q = []
        for parent in parents:
            if parent.left is not None:
                q.append(parent.left)
            if parent.right is not None:
                q.append(parent.right)

    return depths

if __name__ == '__main__':
```

```
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    for depth, list_nodes in enumerate(list_of_depths(root)):
        print('Depth', depth, end=': ')
        for n in list_nodes:
            print(n.data, end=' -> ')
        print('end')
```

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

```
# find the maximum path sum. The path may start and end at any node in the tree.

class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def max_sum_path(root):
    max_sum_path_util.res = -float('inf')

    max_sum_path_util(root)
    return max_sum_path_util.res


def max_sum_path_util(root):
    if root is None:
        return 0
    # find max sum in left and right sub tree
    left_sum = max_sum_path_util(root.left)
    right_sum = max_sum_path_util(root.right)

    # if current node is one of the nodes in the path above for max
    # it can either be alone, or with left sub tree or right sub tree
    max_single = max(max(left_sum, right_sum) + root.data, root.data)
```

```python

    # if the current root itself is considered as top node of the max path
    max_parent = max(left_sum + right_sum + root.data, max_single)

    # store the maximum result
    max_sum_path_util.res = max(max_sum_path_util.res, max_parent)

    # return the max_single for upper nodes calculation
    return max_single


if __name__ == '__main__':
    root = Node(10)
    root.left = Node(2)
    root.right = Node(10)
    root.left.left = Node(20)
    root.left.right = Node(1)
    root.right.right = Node(-25)
    root.right.right.left = Node(3)
    root.right.right.right = Node(4)

    print('max path sum is:', max_sum_path(root))
```

Find minimum depth of a binary tree

```python
# Find minimum depth of a binary tree
from collections import deque


# A Binary Tree node
class Node:
    # Constructor to initialise node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def get_min_depth(root):
    if root is None:
        return 0

```

```python
    queue = deque()
    queue.append((root, 1))

    while queue:
        node, height = queue.popleft()
        if node.left is None and node.right is None:
            return height

        else:
            if node.left is not None:
                queue.append((node.left, height + 1))
            if node.right is not None:
                queue.append((node.right, height + 1))


if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    print(get_min_depth(root))
```

Remove nodes on root to leaf paths of length < K

```python
# Remove nodes on root to leaf paths of length < K


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def remove_path_less_than(root, k):
    return remove_path_util(root, 1, k)


def remove_path_util(root, level, k):
    if root is None:
```

```
        return None

    root.left = remove_path_util(root.left, level+1, k)
    root.right = remove_path_util(root.right, level+1, k)

    if root.left is None and root.right is None and level < k:
        del root
        return None

    return root


def in_order(root):
    if root is None:
        return
    in_order(root.left)
    print(root.data, end=' ')
    in_order(root.right)


if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.left.left.left = Node(7)
    root.right.right = Node(6)
    root.right.right.left = Node(8)
    in_order(root)
    print()
    root = remove_path_less_than(root, 4)
    in_order(root)
```

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

```python
# Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def reverse_alt_levels(root):
    pre_order_rev(root.left, root.right, 0)


def pre_order_rev(root_left, root_right, level):
    # Base case
    if (root_left or root_right) is None:
        return

    # swap the data of nodes if at an alternate level
    if level % 2 == 0:
        root_left.data, root_right.data = root_right.data, root_left.data

    # go to the next level with left of left root and right of right root
    # and vice versa
    pre_order_rev(root_left.left, root_right.right, level+1)
    pre_order_rev(root_left.right, root_right.left, level+1)


def in_order(root):
    if root is None:
        return
    in_order(root.left)
    print(root.data, end=' -> ')
    in_order(root.right)


if __name__ == '__main__':
    root = Node('a')
    root.left = Node('b')
    root.right = Node('c')
    root.left.left = Node('d')
    root.left.right = Node('e')
```

```python
    root.right.left = Node('f')
    root.right.right = Node('g')
    root.left.left.left = Node('h')
    root.left.left.right = Node('i')
    root.left.right.left = Node('j')
    root.left.right.right = Node('k')
    root.right.left.left = Node('l')
    root.right.left.right = Node('m')
    root.right.right.left = Node('n')
    root.right.right.right = Node('o')

    print('Before Reversal:')
    in_order(root)
    print()

    # Call the reverse alternate levels function
    reverse_alt_levels(root)

    print('After Reversal:')
    in_order(root)
    print()
```

Print Nodes in Top View of Binary Tree

```python
# Print Nodes in Top View of Binary Tree
from collections import deque


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def top_view(root):
    if root is None:
        return

    # make an empty queue for BFS
    q = deque()

```

```
    # empty set
    sets = set({})

    # list to store top view keys
    topview = []

    # append root in the queue with horizontal distance as 0
    q.append((root, 0))

    while q:
        # get the element and horizontal distance
        elem, hd = q.popleft()

        # if the hd is seen first time it will be top view
        if hd not in sets:
            topview.append((elem.data, hd))
            sets.add(hd)

        # add left and right child in the queue with hd - 1 and hd + 1
        if elem.left is not None:
            q.append((elem.left, hd - 1))
        if elem.right is not None:
            q.append((elem.right, hd + 1))

    # return the sorted topview on the basis of hd
    return sorted(topview, key=lambda x: x[1])


if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.right = Node(4)
    root.left.right.right = Node(5)
    root.left.right.right.right = Node(6)

    for i in top_view(root):
        print(i[0], end=' ')
```

Implementation of Trie data structure

```
class Node:
    def __init__(self, value=None, isComplete=False):
        self.isComplete = isComplete
        self.children = {}
        self.value = value
        self.isPrefixOf = 0


class Trie:
    def __init__(self):
        self.root = Node()

    def add_word(self, word):
        """
        Add the given word into the trie
        :param word: A String (word) to be added in the trie
        """
        chars = list(word)

        curr_node = self.root

        for ch in chars:
            # The substring till this node will now become a prefix of newly added w\
ord
            curr_node.isPrefixOf += 1

            if ch in curr_node.children:
                curr_node = curr_node.children[ch]
            else:
                new_node = Node(value=ch)
                curr_node.children[ch] = new_node
                curr_node = new_node

        curr_node.isComplete = True

    def search(self, word):
        """
        Searches if the word is present in the Trie or not
        :param word: String (word) to be searched in the trie
        :return: last Node of the searched word if present else None
        """
        chars = list(word)

```

```
        curr_node = self.root

        for ch in chars:
            if ch in curr_node.children:
                curr_node = curr_node.children[ch]
            else:
                return None

        if curr_node.isComplete is True:
            return curr_node

        return None

    def delete(self, word):
        """
        Deletes the given String (word) from the trie
        :param word: Word (String) to be deleted
        :return: True is deleted, False if word not present in the Trie
        """
        chars = list(word)
        n = len(chars)

        val = self._delete(self.root, word)
        return True if val == 1 or val == 0 else False

    def _delete(self, node, chars):
        """
        Recursive Helper function to delete the word and decreement the isPrefix of \
values
        :param node: current node looking at
        :param chars: array of characters to look for
        :return: 1 is word is deleted, 0 if word is deleted and
        """

        # if the chars array is empty
        if len(chars) == 0:
            # check if the word is present in the trie
            if node.isComplete:
                node.isComplete = False

                # check if the word was a prefix of any other words in trie
                # if so, decrement isPrefixOf and return 0, as no deletions are requ\
ired
```

```
                if len(node.children.keys()) > 0:
                    node.isPrefixOf -= 1
                    return 0

                # if word was not a prefix then we need to go up in the trie
                # and find the lowest parent which forms a new word in trie
                return 1
            # if word is not present in the trie
            return -1

        # check if the character is present in current node's children
        if chars[0] in node.children:
            # recursive call for remaining characters in the respective child
            val = self._delete(node.children[chars[0]], chars[1:])

            # if word was found but lowest parent which forms new word is not found
            if val == 1:
                if node.isComplete or len(node.children.keys()) > 1:
                    del node.children[chars[0]]
                    node.isPrefixOf -= 1
                    val = 0
            # if word was found and lowest parent which forms new word was also found
            # simply reduce the isPrefixOf value of the node
            elif val == 0:
                node.isPrefixOf -= 1
            return val

        return -1


trie = Trie()
trie.add_word("anubhav")
trie.add_word("anubshrimal")
trie.add_word("anubhavshrimal")
trie.add_word("data_structures")

if trie.search("anubhav") is not None:
    print("anubhav is present in the Trie")
else:
    print("anubhav is NOT present in the Trie")

trie.delete("anubhav")

```

```
if trie.search("anubhav") is not None:
    print("anubhav is present in the Trie")
else:
    print("anubhav is NOT present in the Trie")

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
[10] https://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

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