**Ex.No: 01 Implementation of Splay Trees**

**Date: 26/02/2024**

Aim:

The aim of this code is to implement a Splay Tree, which is a type of self-adjusting binary search tree. The primary operations include insertion, deletion, search, and tree rotations (left and right rotations).

Algorithm:

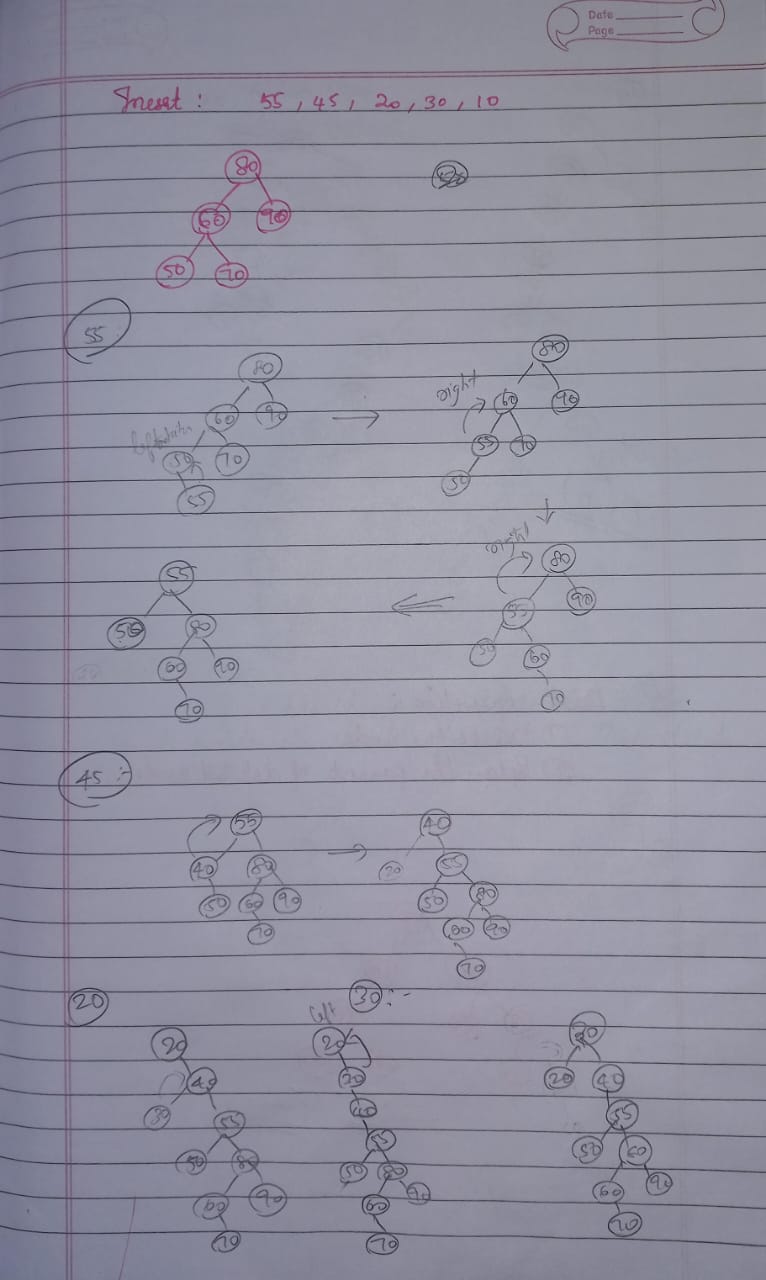
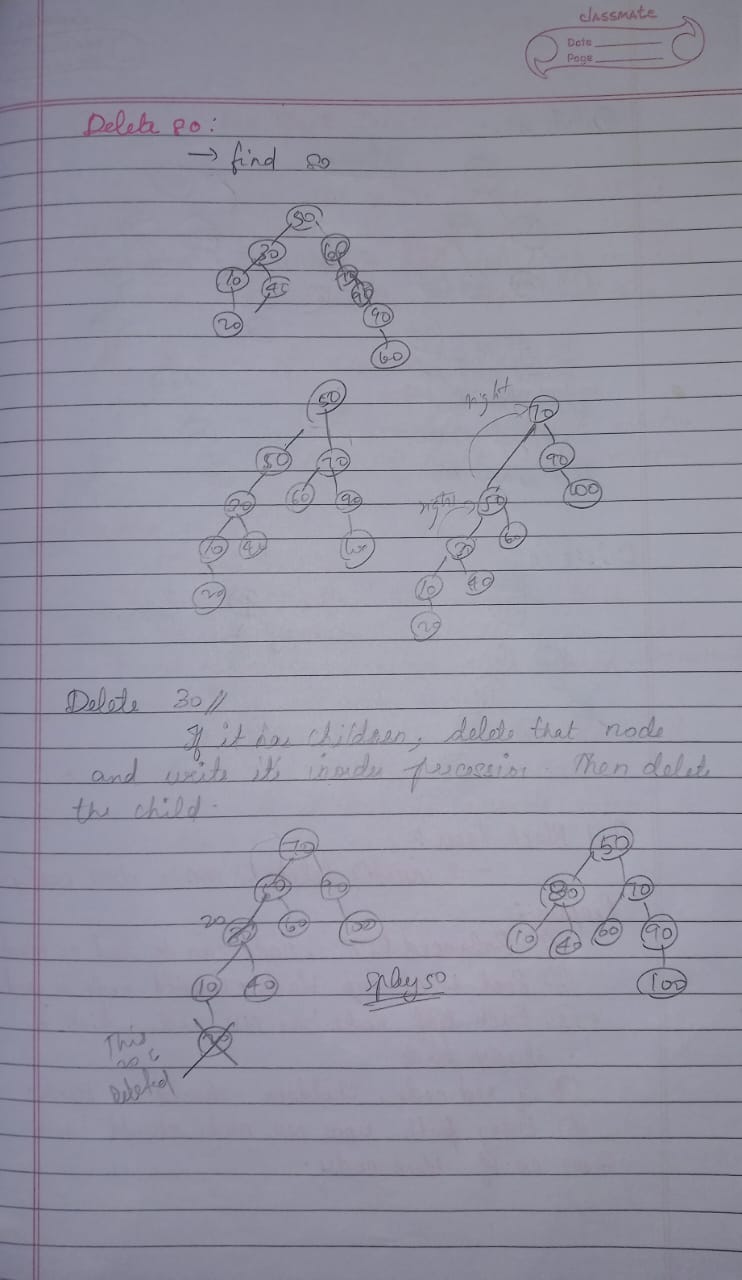
1. Node Class: Defines the structure of a node in the splay tree, including key, left and right children, and parent.

2. ST Class: Implements the splay tree operations:

* Rotations: Includes methods for right and left rotations.
* Splay Operation: Adjusts the tree to bring a specified node to the root through a series of rotations.
* Insertion: Adds a node to the tree and then splays it to the root.
* Search: Looks for a node by key and splays it to the root if found.
* Deletion: Removes a node by key, reconfiguring the tree and adjusting the structure as necessary.
* Traversal: Includes in-order and pre-order traversal methods.

3. Main Function: Demonstrates the insertion and deletion operations along with pre-order traversal to print the tree structure.

Illustration:

Code :

# Node Class

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

# SplayTree class

class SplayTree:

    def \_\_init\_\_(self, root=None):

        self.root = root

    # ZIG-Single right rotation

    def \_right\_rotate(self, node):

        temp = node.left

        node.left = temp.right

        temp.right = node

        return temp

    # ZAG-Single left rotation

    def \_left\_rotate(self, node):

        temp = node.right

        node.right = temp.left

        temp.left = node

        return temp

    # Splay method to perform splay operation while insertion and deletion

    def \_splay(self, node, key):

        if node is None or node.key == key:

            return node

        if key < node.key:

            # no left child

            if node.left is None:

                return node

            # if it is less than node and its left child both

            if key < node.left.key:

                node.left.left = self.\_splay(node.left.left, key)  #returns None because no element

                # perform ZIG-ZIG(2 right rotations) since we need to rotate 2 left elements

                # we perform one now and check if no left element then perform the other

                node = self.\_right\_rotate(node)

            elif key > node.left.key:

                # ZAG-ZIG rotation

                node.left.right = self.\_splay(node.left.right, key)#returns None because no current element we later add it in insert method

                node = self.\_left\_rotate(node)

            # finally we perform the second right rotation

            if node.left is not None:

                return self.\_right\_rotate(node)

            else:

                return node

        else: # key > node.key

            if node.right is None:

                return node

            if key > node.right.key:

                node.right.right = self.\_splay(node.right.right, key)

                # ZAG-ZAG rotation

                node = self.\_left\_rotate(node)

            elif key < node.right.key:

                node.right.left = self.\_splay(node.right.left, key)

                # ZIG-ZAG rotation

                node = self.\_right\_rotate(node)

            # finally we perform the second left rotation

            if node.right is not None:

                return self.\_left\_rotate(node)

            else:

                return node

    # Inserting a value (INSERT)

    def insert(self, key):

        if self.root is None:

            self.root = Node(key)

        else:

            self.root = self.\_insert(self.root, key)

    def \_insert(self, node, key):  # here Node is the root and key is the value to be inserted

        if node is None:

            return Node(key)

        node = self.\_splay(node, key)

        if key < node.key:

            new\_node = Node(key)

            new\_node.left = node.left

            new\_node.right = node

            node.left = None  # remember this alone don't forget to make it None

            return new\_node

        elif key > node.key:

            new\_node = Node(key)

            new\_node.right = node.right

            new\_node.left = node

            node.right = None

            return new\_node

        else:

            # key already exists we just splay the node to the root

            # return the splayed node

            return node

    # Deleting a value (DELETE)

    def delete(self, key):

        if self.root is None:

            return None

        self.root = self.\_delete(self.root, key)

    def \_delete(self, node, key):

        node = self.\_splay(node, key)

        if key != node.key:

            return node

        if node.left is None:

            return node.right

        else:

            right\_subtree = node.right  # Store the right subtree in a temp variable

            node = node.left

            node = self.\_splay(node, key)

            node.right = right\_subtree

            return node

    # Searching a value (SEARCH)

    def search(self, key):

        self.root = self.\_splay(self.root, key)

        return self.root.key == key if self.root else False

    def pre\_order\_traversal(self, node):

        if node is None:

            return

        print(node.key, end=" ")

        self.pre\_order\_traversal(node.left)

        self.pre\_order\_traversal(node.right)

# DRIVER CODE

if \_\_name\_\_ == "\_\_main\_\_":

    splay\_tree = SplayTree()

    splay\_tree.insert(10)

    splay\_tree.insert(5)

    splay\_tree.insert(15)

    splay\_tree.insert(20)

    splay\_tree.insert(25)

    print("search result :",splay\_tree.search(25))

    print("Before deletion:")

    print(splay\_tree.search(15))

    splay\_tree.delete(15)

    print("After deletion:")

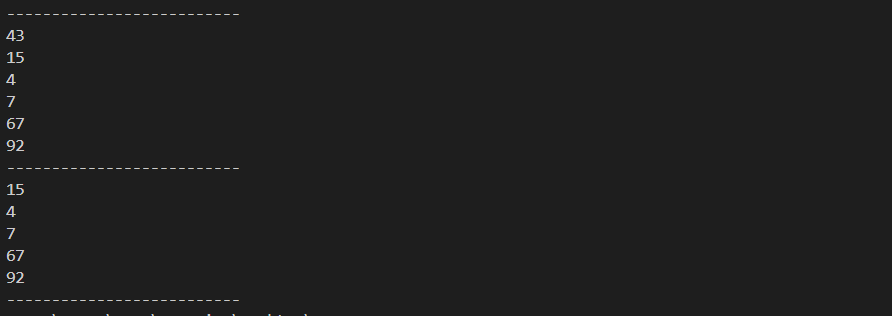
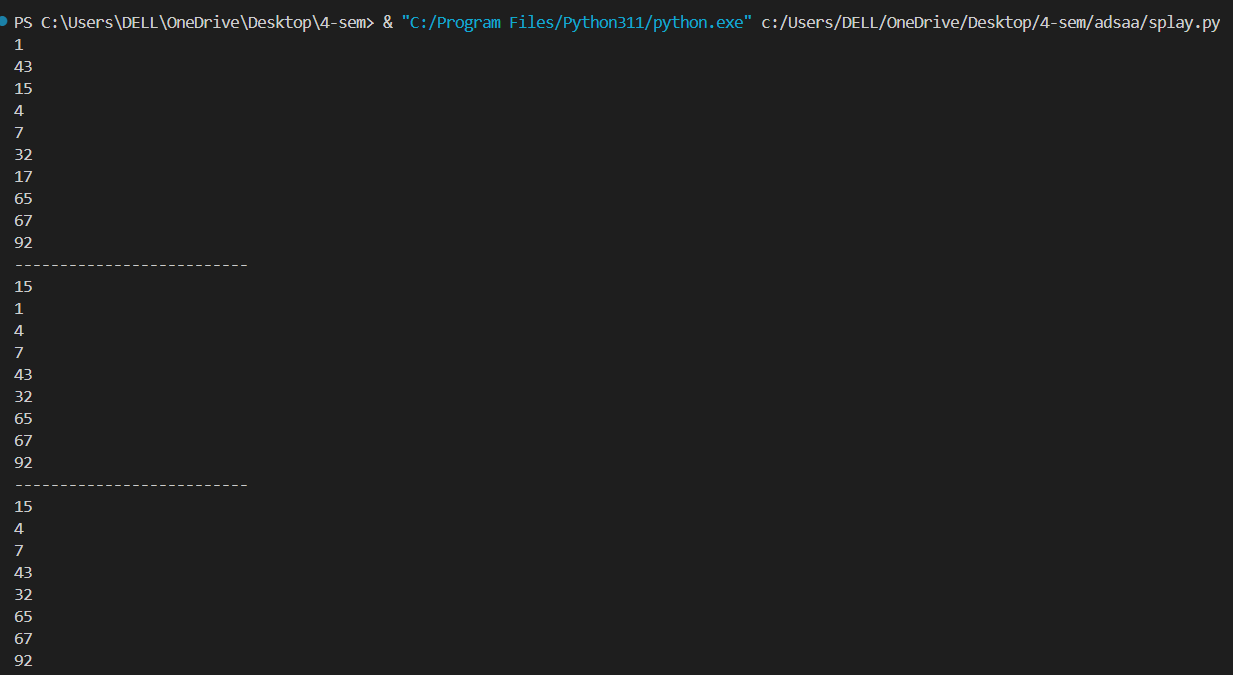
    print(splay\_tree.search(15))

    #preorder traversal

    print("Pre-order traversal:")

    splay\_tree.pre\_order\_traversal(splay\_tree.root)

Output:



Time Complexity Summary

* Insertion of a single node: O(log n) amortized
* Insertion of a list of m elements: O(m log n)
* Search: O(log n) amortized
* Splay: O(log n) amortized
* Rotation: O(1)
* Deletion: O(log n) amortized
* In-order traversal: O(n)
* Pre-order traversal: O(n)

The splay tree's performance for insertion, search, and deletion operations is efficient, with an amortized time complexity of O(log n), making it well-suited for dynamic set operations where the access pattern exhibits temporal locality.

Result :

The splay tree with insert and delete functionalities were implemented successfully.

**Ex.No: 02 Implementation of B+ Trees**

**Date: 04/03/2024**

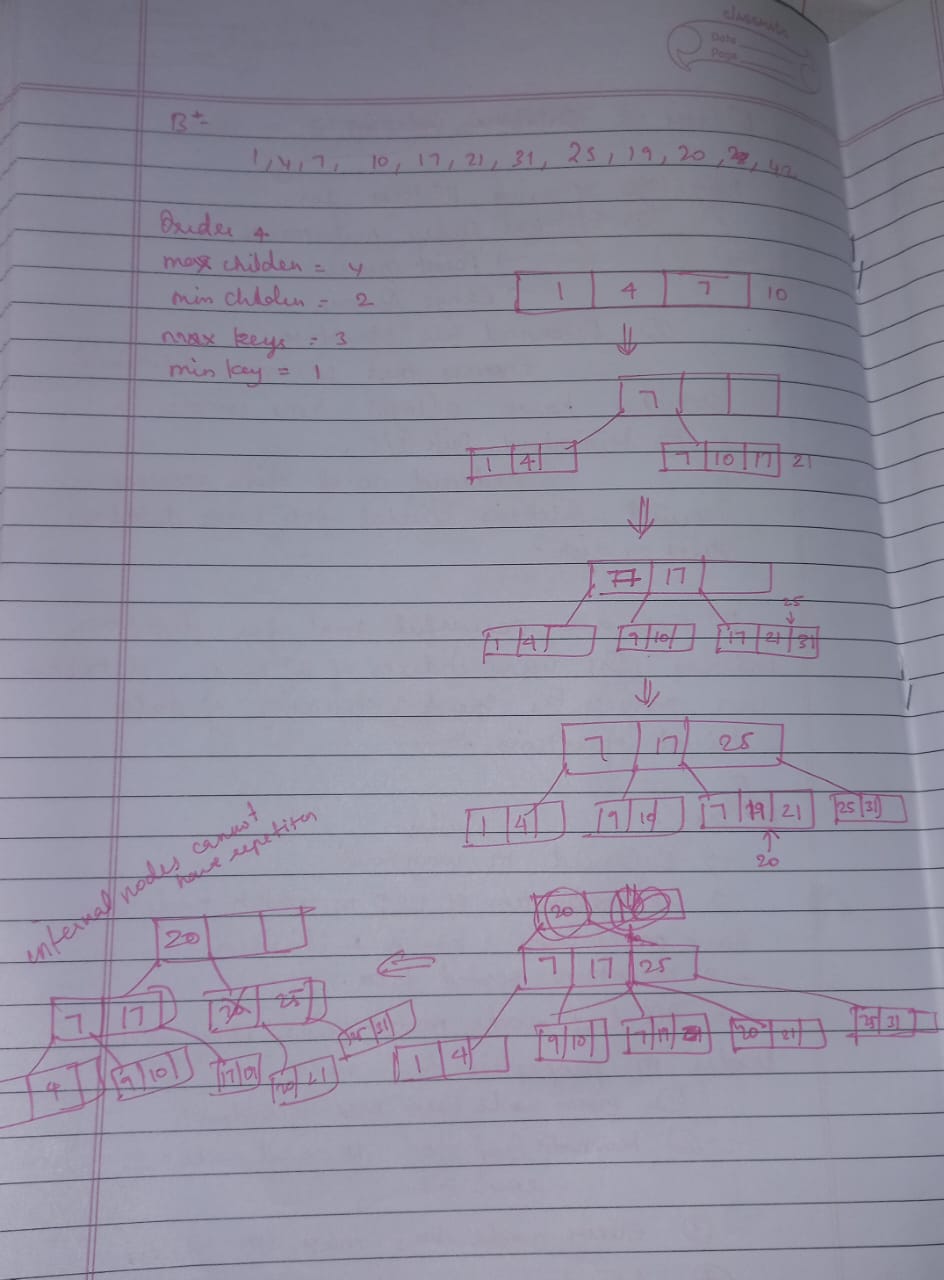
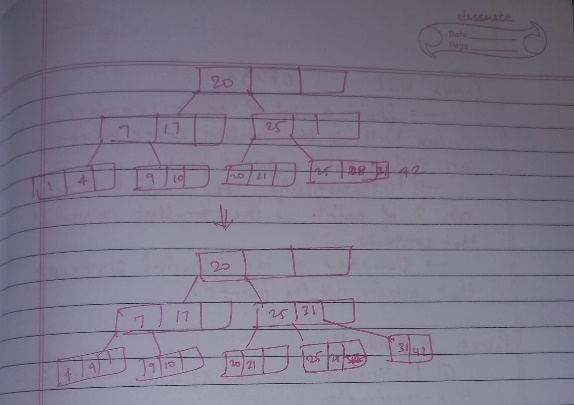
Aim:

To implement a B+ Tree, a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations, as well as range queries.

Algorithm:

* BPlusTree Class: Initializes the tree with a specified degree and sets the root to a new leaf node.
* Insert in BPlusTree: Insert a key-value pair into the root. If the root node becomes full, split it and create a new root.
* Insert in LeafNode: Insert the key in the appropriate position within the leaf node. If the leaf node becomes full, it will be split by its parent (an internal node).
* Delete in BPlusTree: Delete a key from the root. If the root becomes empty and has only one child, that child becomes the new root.
* Delete in LeafNode: Remove the key from the leaf node. Handle any underflow by borrowing or merging nodes as needed, managed by the parent internal node.
* Search in BPlusTree: Start the search from the root and traverse down to the leaf node where the key may reside.
* Search in LeafNode: Locate the key within the leaf node and return the associated value.
* Range Query in BPlusTree: Find the starting leaf node for the given range. Traverse through the linked leaf nodes, collecting values for keys within the specified range.
* Split Child in InternalNode: When a child node becomes full, split it into two nodes and adjust the parent node to accommodate the split.
* Handle Underflow in InternalNode: When a child node becomes underfull, rebalance by borrowing keys from sibling nodes or merging nodes if necessary. Adjust the parent node accordingly.

Illustration :

Code :

class Node:

    def \_\_init\_\_(self, degree, is\_leaf=True):

        self.degree = degree

        self.keys = []

        self.children = []

        self.is\_leaf = is\_leaf

        self.next=None

class BPlusTree:

    def \_\_init\_\_(self, degree):

        self.degree = degree

        self.root = Node(degree)

    def insert(self, key):

        root = self.root

        if len(root.keys) == (2 \* self.degree) - 1:

            new\_root = Node(self.degree, is\_leaf=False)

            new\_root.children.append(root)

            self.\_split\_child(new\_root, 0, root)

            i = 0

            if new\_root.keys[0] < key:

                i += 1

            self.\_insert\_non\_full(new\_root.children[i], key)

            self.root = new\_root

        else:

            self.\_insert\_non\_full(root, key)

    def \_insert\_non\_full(self, node, key):

        i = len(node.keys) - 1

        if node.is\_leaf:

            node.keys.append(None)

            while i >= 0 and key < node.keys[i]:

                node.keys[i + 1] = node.keys[i]

                i -= 1

            node.keys[i + 1] = key

        else:

            while i >= 0 and key < node.keys[i]:

                i -= 1

            i += 1

            if len(node.children[i].keys) == (2 \* self.degree) - 1:

                self.\_split\_child(node, i, node.children[i])

                if key > node.keys[i]:

                    i += 1

            self.\_insert\_non\_full(node.children[i], key)

    def \_split\_child(self, parent, i, child):

        new\_child = Node(self.degree, is\_leaf=child.is\_leaf)

        parent.children.insert(i + 1, new\_child)

        parent.keys.insert(i, child.keys[self.degree - 1])

        new\_child.keys = child.keys[self.degree:]

        child.keys = child.keys[:self.degree - 1]

        if not child.is\_leaf:

            new\_child.children = child.children[self.degree:]

            child.children = child.children[:self.degree]

    def search(self, key):

        return self.\_search(self.root, key)

    def \_search(self, node, key):

        i = 0

        while i < len(node.keys) and key > node.keys[i]:

            i += 1

        if i < len(node.keys) and key == node.keys[i]:

            return True

        if node.is\_leaf:

            return False

        return self.\_search(node.children[i], key)

    def traverse(self):

        if self.root:

            self.\_traverse(self.root)

            print()

    def \_traverse(self, node):

        i = 0

        while i < len(node.keys):

            if not node.is\_leaf:

                self.\_traverse(node.children[i])

            print(node.keys[i], end=" ")

            i += 1

        if not node.is\_leaf:

            self.\_traverse(node.children[i])

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    b\_tree = BPlusTree(degree=3)  # Example degree

    keys\_to\_insert = [1,2,3,4,5,6,7,8,9,10,11,12,13,14]  # Insertion

    for key in keys\_to\_insert:

        b\_tree.insert(key)

    # Searching

    search\_keys = [1,2,3,4,5,8,90,43]

    for key in search\_keys:

        if b\_tree.search(key):

            print(f"Key {key} found in the B+ tree.")

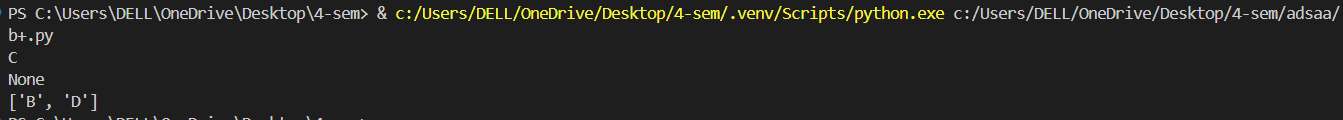
        else:

            print(f"Key {key} not found in the B+ tree.")

    print("Traversal of tree constructed is")

    b\_tree.traverse()

Output:



Time Analysis:

 Insertion:

* Finding the correct leaf node: O(log\_d n), where d is the degree and n is the number of keys.
* Inserting into the leaf node: O(d).
* Splitting nodes and propagating splits up the tree: O(d log\_d n).
* Total insertion time complexity: O(d log\_d n).

 Deletion:

* Finding the correct leaf node: O(log\_d n).
* Deleting from the leaf node: O(d).
* Handling underflows and merging nodes: O(d log\_d n).
* Total deletion time complexity: O(d log\_d n).

 Search:

* Finding the correct leaf node: O(log\_d n).
* Searching within the leaf node: O(d).
* Total search time complexity: O(d log\_d n).

 Range Query:

* Finding the starting leaf node: O(log\_d n).
* Traversing through leaf nodes and collecting values: O(k) where k is the number of elements in the range.
* Total range query time complexity: O(log\_d n + k).

Result :

The B+ tree was successfully implemented.

**Ex.No: 03 Implementation of Space Search Algorithm**

**Date: 11/03/2024**

Aim:

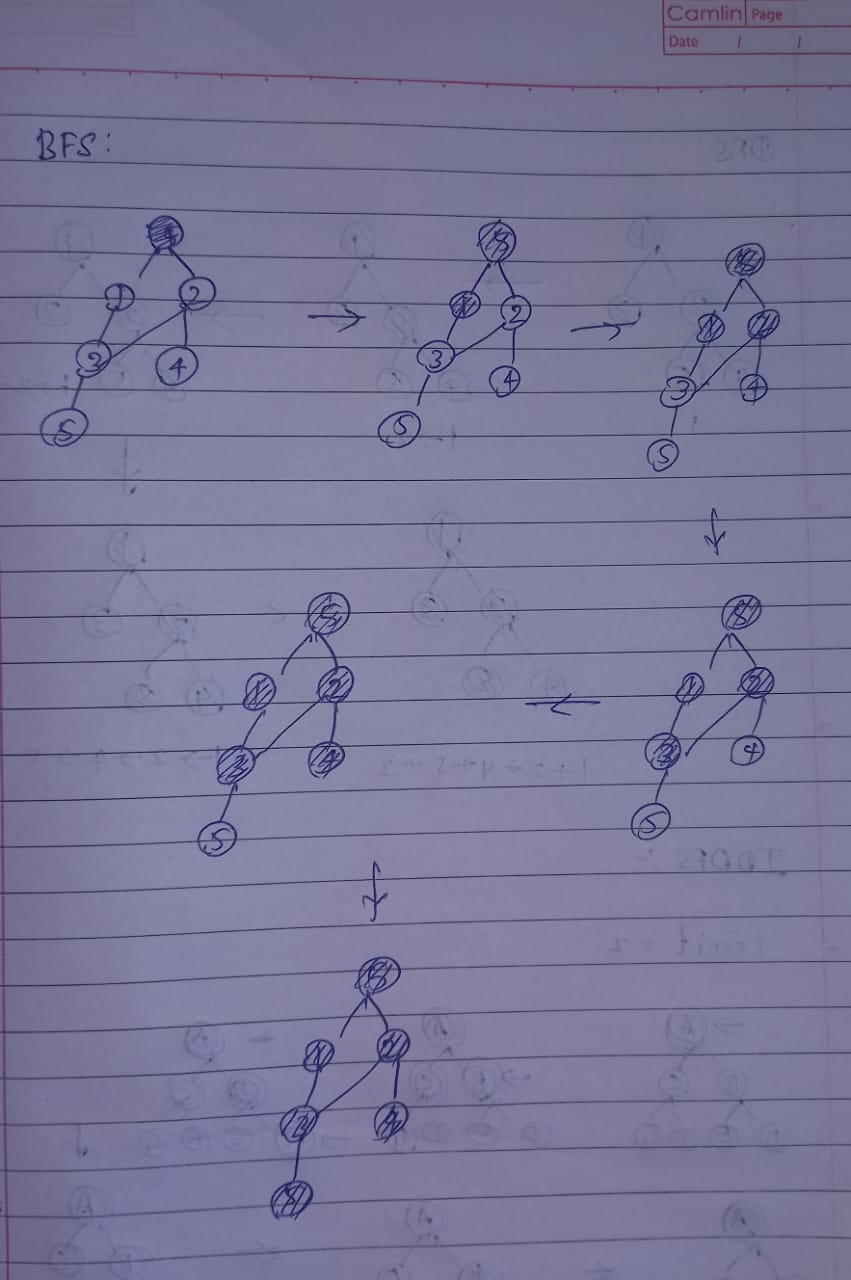
To implement the state space search algorithms like breadth first search, depth first search and iterative deepening.

1. Breadth-first Search

Algorithm :

* Create an empty list visited to keep track of visited nodes.
* Create an empty queue for nodes to visit next.
* Append the starting node to visited and queue.
* While the queue is not empty:
  + Dequeue the front node (m) from the queue and print it.
  + For each neighbor of node m:
    - If the neighbor has not been visited:
      * Append the neighbor to visited.
      * Enqueue the neighbor in queue.
* End when all nodes in the queue have been visited.

Illustration :



Code :

graph = {  '5' : ['3','7'],

  '3' : ['2', '4'],

  '7' : ['8'],

  '2' : [],

  '4' : ['8'],

  '8' : []

}

visited = [] # List for visited nodes.

queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS

  visited.append(node)

  queue.append(node)

  while queue:          # Creating loop to visit each node

    m = queue.pop(0)

    print (m, end = " ")

    for neighbour in graph[m]:

      if neighbour not in visited:

        visited.append(neighbour)

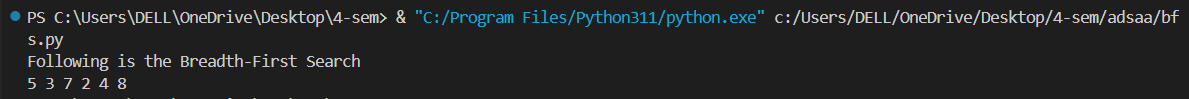
        queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '5')    # function calling

Output:

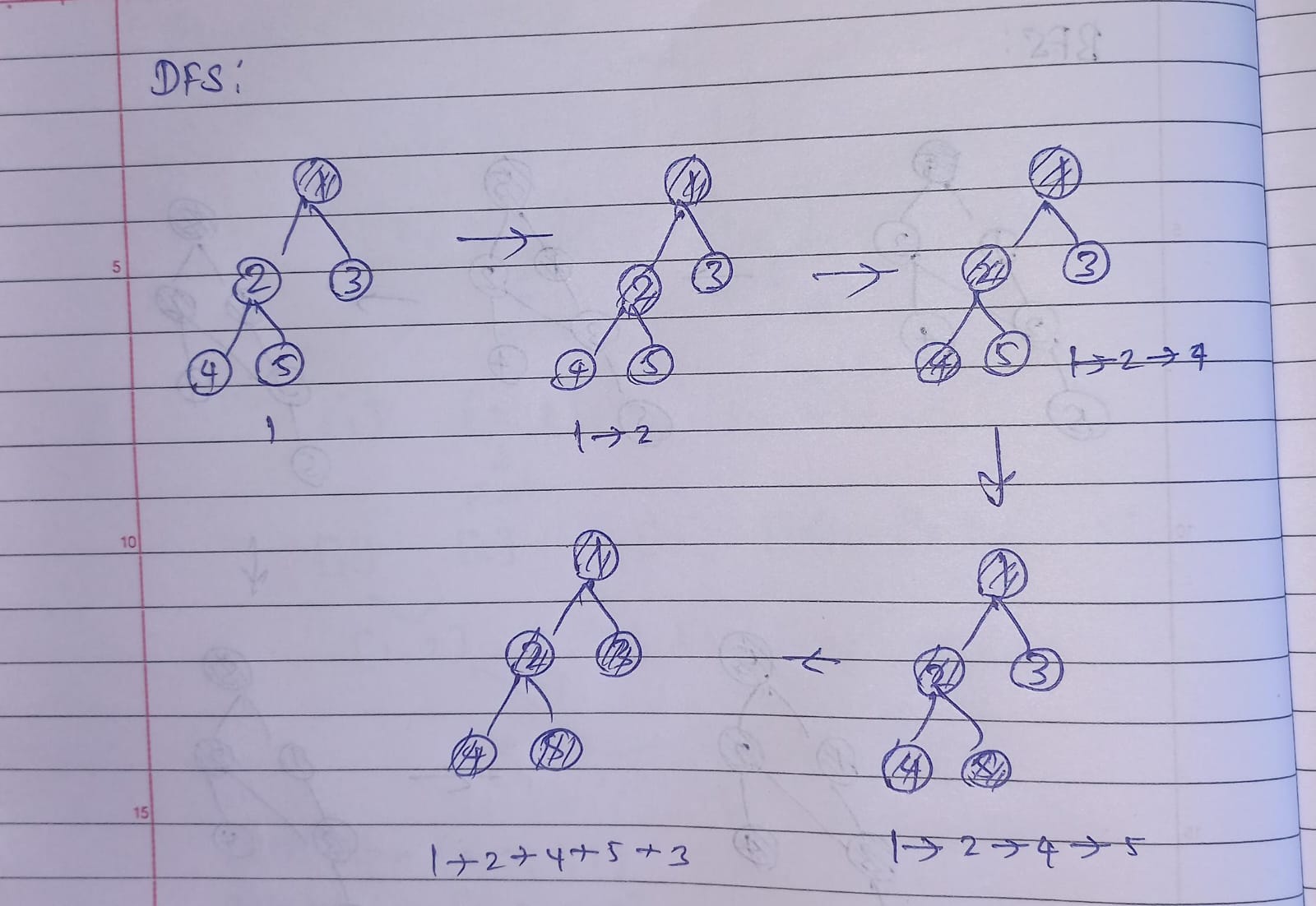


1. Depth-First Search

Algorithm :

* Visited is an empty set that will store nodes as they are visited.
* The starting node '5' is checked. If it has not been visited:
  + The node is printed.
  + The node is added to visited.
* For each neighbor of the current node:
  + If the neighbor has not been visited:
    - The DFS function is called recursively on the neighbor.
* The nodes are printed in the order they are visited.

Illustration :



Code :

graph = {

  '5' : ['3','7'],

  '3' : ['2', '4'],

  '7' : ['8'],

  '2' : [],

  '4' : ['8'],

  '8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs

    if node not in visited:

        print (node)

        visited.add(node)

        for neighbour in graph[node]:

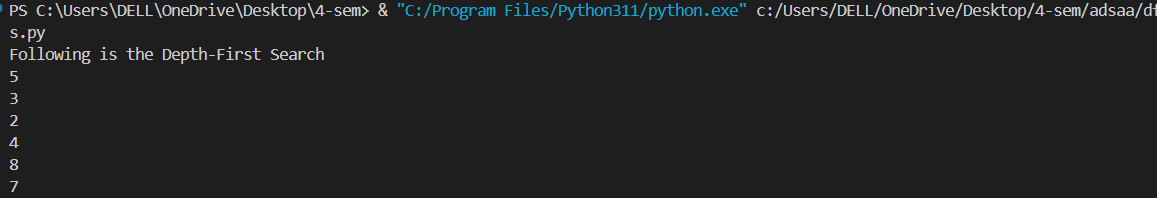
            dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

Output:

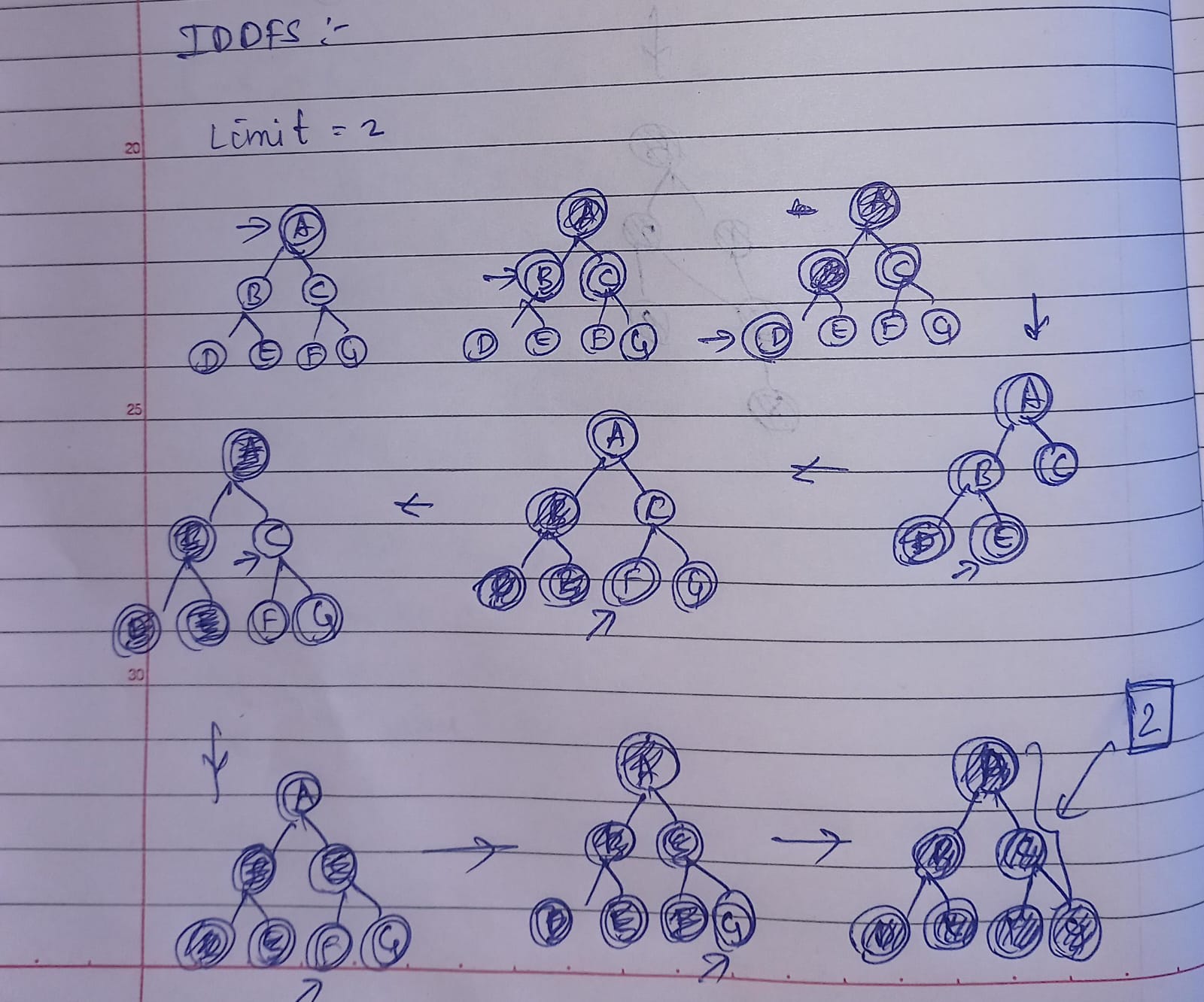


1. Iterative Deepening

Algorithm :

* Define the graph as an adjacency list.
* Use a stack to keep track of nodes and their depth.
* While the stack is not empty:
  + Pop a node and its depth from the stack.
  + If the node is the target and within the depth limit, return True.
  + If the node is within the depth limit:
    - Add it to the visited set.
    - Push all its unvisited neighbors onto the stack with incremented depth.
* Loop from depth 0 to the maximum depth:
  + Call DLS with the current depth limit.
  + If the target is found, return True.
* Return False if the target is not found within the maximum depth.

Illustration :



Code :

def dls(graph, start, target, limit):

    stack = [(start, 0)]

    visited = set()

    while stack:

        vertex, depth = stack.pop()

        if depth <= limit:

            if vertex == target:

                print(f"Target {target} found at depth {depth}")

                return True

            visited.add(vertex)

            # Add neighbors to the stack if not visited

            stack.extend((neigh, depth + 1) for neigh in graph[vertex] if neigh not in visited)

    return False

def iddfs(graph, start, target, max\_depth):

    for limit in range(max\_depth + 1):

        if dls(graph, start, target, limit):

            return True

    return False

graph = {

    '5': ['3', '7'],

    '3': ['2', '4'],

    '7': ['8'],

    '2': [],

    '4': ['8'],

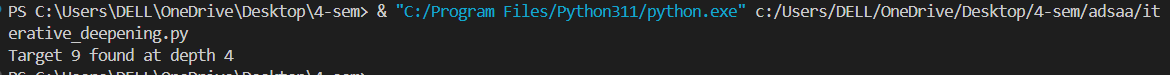
    '8': ['10'],

    '10': ['9']

}

iddfs(graph, '5', '9', 4)

Output:



Result:

The state space search for bfd,dfs,iddfs were implemented and executed successfully.s

**Ex.No: 04 Implementation of Divide and Conquer Algorithm**

**Date: 18/03/2024**

Aim:

To implement Divide and Conquer algorithms like merge sort, quick sort and closest pair problem.

1. Merge Sort

Algorithm :

* Base Case:

If the array has one or zero elements, it is already sorted. Return the array.

* Divide:

Split the array into two halves.

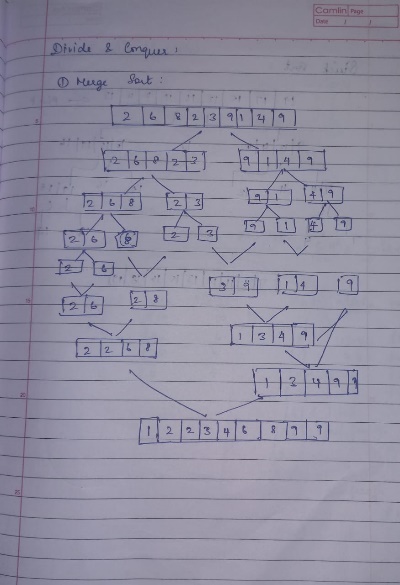
* Conquer:

Recursively sort each half.

* Merge:

Merge the two sorted halves into a single sorted array.

Illustration :



Code :

def msort(arr):

    if len(arr) <= 1:

        return arr

    mid = len(arr) // 2

    left = msort(arr[:mid])

    right = msort(arr[mid:])

    return merge(left, right)

def merge(left, right):

    result = []

    i = j = 0

    while i < len(left) and j < len(right):

        if left[i] <= right[j]:

            result.append(left[i])

            i += 1

        else:

            result.append(right[j])

            j += 1

    result += left[i:]

    result += right[j:]

    return result

if \_\_name\_\_ == "\_\_main\_\_":

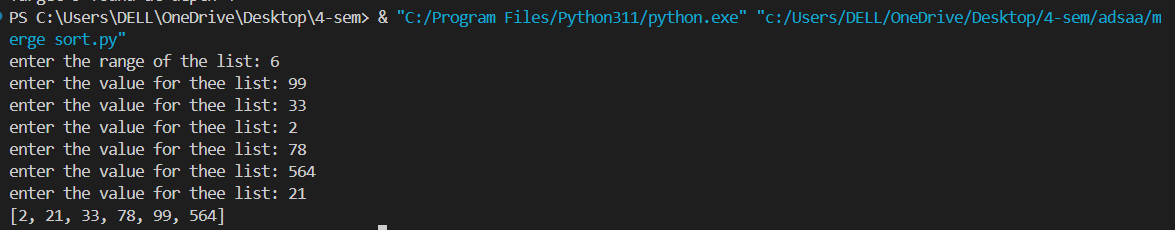
    arr=[]

    length=int(input("enter the range of the list: "))

    for i in range(length):

        arr.append(int(input("enter the value for thee list: ")))

    print(msort(arr))

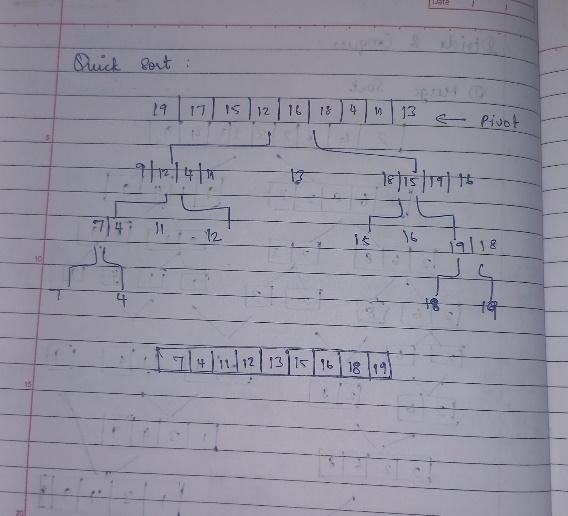
Output :

1. Quick sort

Algorithm :

* If the array has one or zero elements, it is already sorted. Return the array.
* Select the first element of the array as the pivot.
* Create two sub-arrays:
* lesser\_than\_pivot: Elements less than or equal to the pivot.
* greater\_than\_pivot: Elements greater than the pivot.
* Recursively Sort:
* Apply Quick Sort to the lesser\_than\_pivot and greater\_than\_pivot sub-arrays.
* Combine:
* Concatenate the sorted lesser\_than\_pivot, the pivot, and the sorted greater\_than\_pivot.

Illustration :



Code :

def quicksort(arr):

    if len(arr)<=1:

        return arr

    else:

        pivot=arr[0]

        lesser\_than\_pivot=[x for x in arr[1:] if x <= pivot]

        greater\_than\_pivot=[x for x in arr[1:] if x > pivot]

        return quicksort(lesser\_than\_pivot)+[pivot]+quicksort(greater\_than\_pivot)

arr=[8,1,3,45,888,90]

print(quicksort(arr))

Output:



1. Closest pair problem

Algorithm :

 Define Point Class:

* Create a Point class with x and y coordinates.

 Distance Calculation:

* Define a function dist to compute the Euclidean distance between two points.

 Brute Force Method:

* bruteForce(P, n): Find the minimum distance between all pairs of points in a small set of points P.

 Strip Closest:

* stripClosest(strip, size, d): Find the closest points in a vertical strip of width 2d, sorted by the y-coordinate.

 Recursive Closest Util:

* closestUtil(P, n): Recursively find the smallest distance by:
  + Dividing the set into two halves.
  + Finding the smallest distance in both halves (dl and dr).
  + Finding the smallest distance in the strip around the midpoint.

 Closest Pair Function:

* closest(P, n): Sort the points by x-coordinate and call closestUtil

Code : import math

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

def compareX(a, b):

    p1 = a

    p2 = b

    return (p1.x - p2.x)

def compareY(a, b):

    p1 = a

    p2 = b

    return (p1.y - p2.y)

def dist(p1, p2):

    return math.sqrt((p1.x - p2.x)\*(p1.x - p2.x) + (p1.y - p2.y)\*(p1.y - p2.y))

def bruteForce(P, n):

    min\_dist = float("inf")

    for i in range(n):

        for j in range(i+1, n):

            if dist(P[i], P[j]) < min\_dist:

                min\_dist = dist(P[i], P[j])

    return min\_dist

def min(x, y):

    return x if x < y else y

def stripClosest(strip, size, d):

    min\_dist = d

    strip = sorted(strip, key=lambda point: point.y)

    for i in range(size):

        for j in range(i+1, size):

            if (strip[j].y - strip[i].y) >= min\_dist:

                break

            if dist(strip[i], strip[j]) < min\_dist:

                min\_dist = dist(strip[i], strip[j])

    return min\_dist

def closestUtil(P, n):

    if n <= 3:

        return bruteForce(P, n)

    mid = n//2

    midPoint = P[mid]

    dl = closestUtil(P, mid)

    dr = closestUtil(P[mid:], n - mid)

    d = min(dl, dr)

    strip = []

    for i in range(n):

        if abs(P[i].x - midPoint.x) < d:

            strip.append(P[i])

    return min(d, stripClosest(strip, len(strip), d))

def closest(P, n):

    P = sorted(P, key=lambda point: point.x)

    return closestUtil(P, n)

if \_\_name\_\_ == "\_\_main\_\_":

    P = [Point(x=2, y=3), Point(x=12, y=30),

        Point(x=40, y=50), Point(x=5, y=1), Point(x=12, y=10), Point(x=3, y=4)]

    n = len(P)

    print("The smallest distance is", closest(P, n))

Output :



Result:

The merge sort, quick sort and closest pair problem are implemented using divide and conquer algorithm successfully.

**Ex.No: 05 Implementation of Huffman Coding**

**Date: 25/03/2024**

Aim:

To implement Huffman coding using greedy technique.

Algorithm :

 Initialization:

* Create a node class to store frequency, symbol, left and right children, and Huffman code.
* Initialize a list of nodes for each character with its frequency.

 Create Min-Heap:

* Insert all nodes into a min-heap (priority queue) based on their frequency.

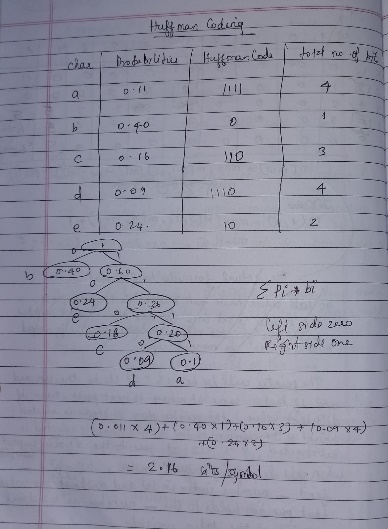
 Build Huffman Tree:

* While there is more than one node in the heap:
  + Extract the two nodes with the smallest frequencies.
  + Create a new internal node with these two nodes as children.
  + Assign a directional value (0 or 1) to the left and right children.
  + Insert the new node back into the heap.

 Generate Huffman Codes:

* Traverse the Huffman Tree to assign and print Huffman codes for each character.

Illustration :



Code :

import heapq

class node:

    def \_\_init\_\_(self, freq, symbol, left=None, right=None):

        self.freq = freq

        self.symbol = symbol

        self.left = left

        self.right = right

        self.huff = ''

    def \_\_lt\_\_(self, nxt):

        return self.freq < nxt.freq

def printNodes(node, val=''):

    # huffman code for current node

    newVal = val + str(node.huff)

    if(node.left):

        printNodes(node.left, newVal)

    if(node.right):

        printNodes(node.right, newVal)

    if(not node.left and not node.right):

        print(f"{node.symbol} -> {newVal}")

chars = ['a', 'b', 'c', 'd', 'e', 'f']

freq = [5, 9, 12, 13, 16, 45]

freq\_prob = [freq[i]/sum(freq) for i in range(len(freq))]

print(freq\_prob)

nodes = []

for x in range(len(chars)):

    heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:

    # sort all the nodes in ascending order based on their frequency

    left = heapq.heappop(nodes)

    right = heapq.heappop(nodes)

    # assign directional value to these nodes

    left.huff = 0

    right.huff = 1

    # combine the 2 smallest nodes to create

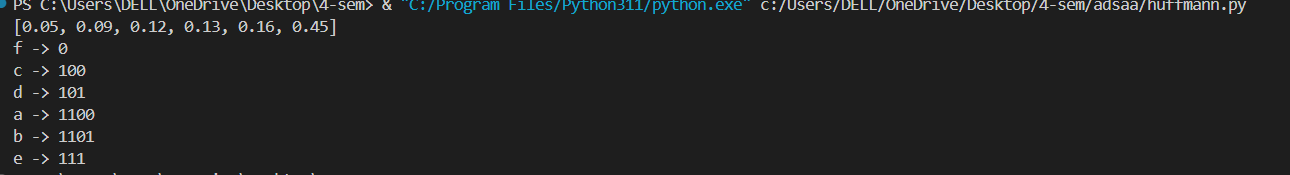
    # new node as their parent

    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

    heapq.heappush(nodes, newNode)

printNodes(nodes[0])

Output :



Result :

The Huffman coding using greedy technique is successfully implemented and executed.

**Ex.No: 06 Implementation of Disjoint sets and Kruskal Algorithm**

**Date: 01/04/2024**

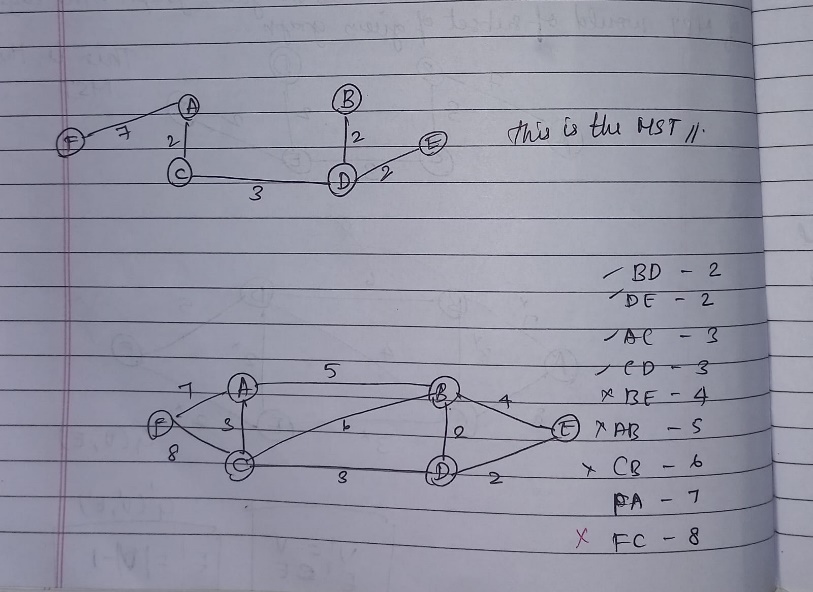
Aim:

The aim of Kruskal's algorithm is to find the minimum spanning tree (MST) of a connected, undirected graph. The MST is a subset of the graph's edges that connects all vertices with the minimum possible total edge weight, without forming any cycles.

Algorithm:

1. Initialization:
   * Create a list of all edges in the graph, each represented by a tuple containing two vertices and the edge weight.
   * Create a set for each vertex, initially containing only the vertex itself.
2. Sorting:
   * Sort all the edges in non-decreasing order based on their weights.
3. MST Construction:
   * Initialize an empty list to store the edges of the MST.
   * Iterate through the sorted edge list and for each edge:
     + Check if the vertices of the edge belong to different sets.
     + If they do, add the edge to the MST and union the sets containing these vertices.
     + If they belong to the same set, ignore the edge to avoid cycles.
4. Termination:
   * The algorithm terminates when the MST contains V−1V-1V−1 edges, where VVV is the number of vertices in the graph.

Illustration:



Code :

# kruskal's minimum spanning tree

class Graph:

    def \_\_init\_\_(self):

        self.edges = [] # list of tuples, each tuple stating vertex1, vertex2 and cost

        self.vertices = set()

    def add\_edge(self, v1, v2, w):

        self.edges.append((v1, v2, w))

        self.vertices.add(v1) # by default you cannot add duplicate data

        self.vertices.add(v2)

    def find\_set(self, v, sets):

        for s in sets:

            if v in s:

                return s

        return None

    def union(self, s1, s2):

        return s1.union(s2)

    def kruskals(self):

        sets = [{v} for v in self.vertices]

        sorted\_edges = sorted(self.edges, key=lambda x: x[2]) # sort the edges based on their cost

        mst = []

        for edge in sorted\_edges:

            v1, v2, w = edge # tuple unpacking

            set\_v1 = self.find\_set(v1, sets)

            set\_v2 = self.find\_set(v2, sets)

            if set\_v1 != set\_v2: # if both the vertices do not belong to the same set

                mst.append((v1, v2, w))

                new\_set = self.union(set\_v1,set\_v2)

                sets.remove(set\_v1)

                sets.remove(set\_v2)

                sets.append(new\_set)

        return mst

    def print\_graph(self):

        print("Original Graph:")

        for edge in self.edges:

            print(f"Edge: {edge[0]} - {edge[1]}, Weight: {edge[2]}")

    def print\_mst(self, mst):

        print("\nMinimum Spanning Tree (MST):")

        for edge in mst:

            print(f"Edge: {edge[0]} - {edge[1]}, Weight: {edge[2]}")

# Create a graph

graph = Graph()

# Add edges with vertices and weights

graph.add\_edge('A', 'B', 4)

graph.add\_edge('A', 'C', 2)

graph.add\_edge('B', 'C', 5)

graph.add\_edge('B', 'D', 10)

graph.add\_edge('C', 'D', 3)

graph.add\_edge('C', 'D', 2)

# Print the original graph

graph.print\_graph()

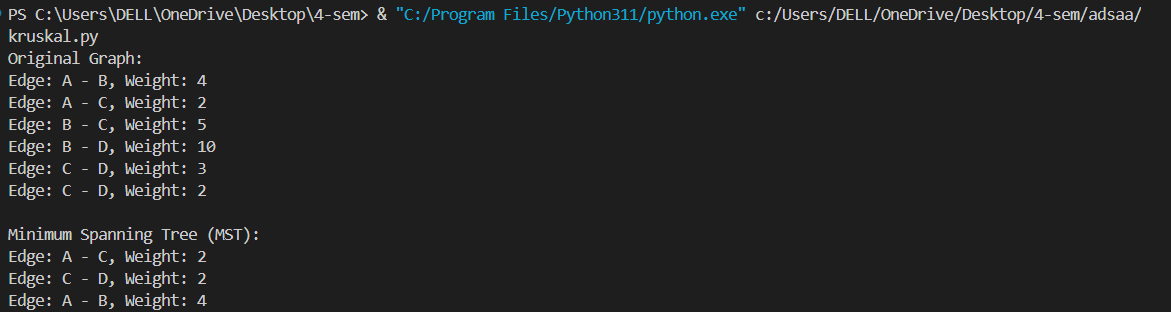
# Find the minimum spanning tree

mst = graph.kruskals()

# Print the minimum spanning tree (MST)

graph.print\_mst(mst)

Output:



Result :

The kruskal algorithm using greedy technique is successfully implemented and executed.

**Ex.No: 07 Implementation of Dynamic algorithms: Binomial**

**Co-efficient,BellmanFord Algorithm**

1. Binomial Co-efficient:

**Date: 08/04/2024**

Aim

The aim of the dynamic programming approach for calculating the Binomial Coefficient C(n,k)C(n, k)C(n,k) is to efficiently compute the number of ways to choose k elements from a set of n elements without recalculating overlapping subproblems.

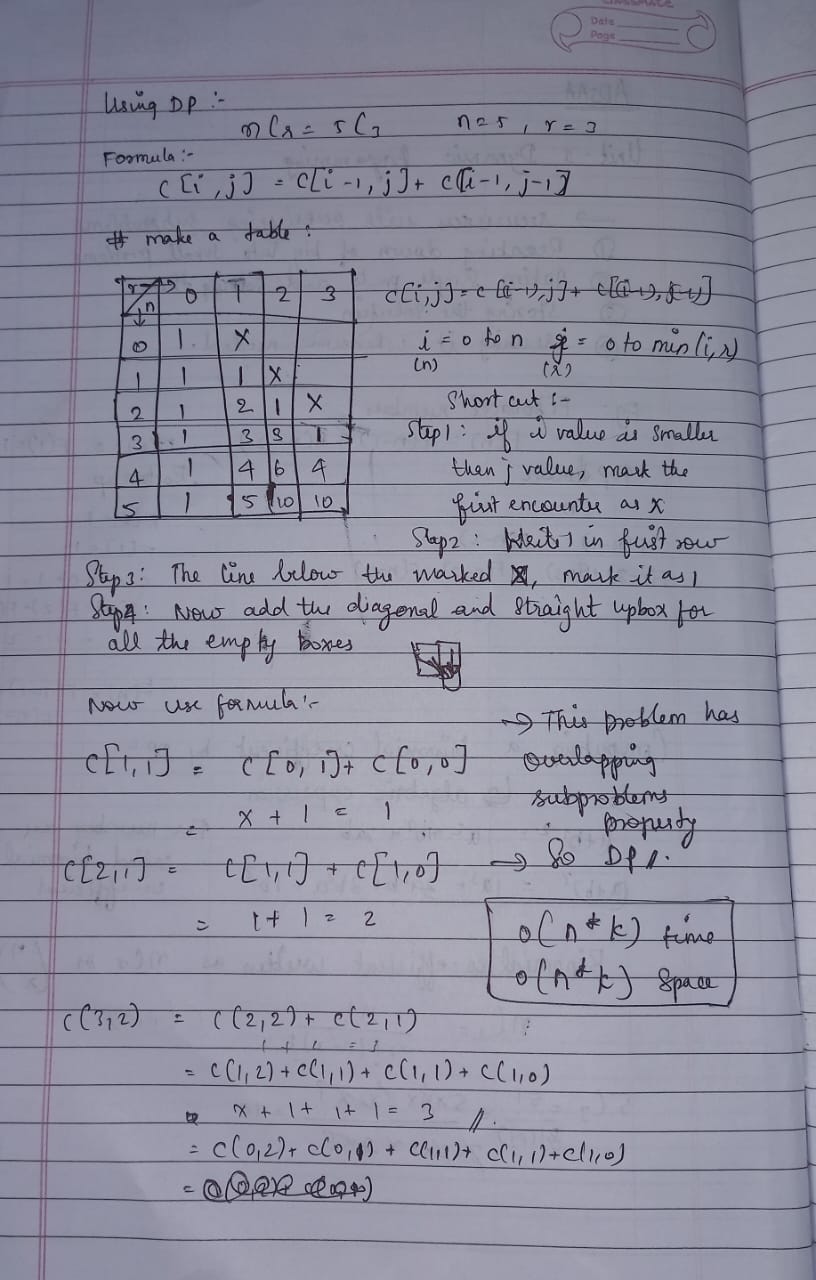
Algorithm :

* The binomial coefficient C(n,k)C(n, k)C(n,k) can be defined as:
  + - C(n,k)=n!/k!(n−k
* Using dynamic programming, we avoid redundant calculations by storing intermediate results. The recursive relation for C(n,k) is:
  + - C(n,k)=C(n−1,k−1)+C(n−1,k)
* The base cases are: C(n,0)=C(n,n)=1

Steps:

1. Create a 2D array dp where dp[i][j] will store the value of C(i,j)C(i, j)C(i,j).
2. Initialize the base cases:
   * dp[i][0] = 1 for all iii because C(i,0)=1C(i, 0) = 1C(i,0)=1.
   * dp[i][i] = 1 for all iii because C(i,i)=1C(i, i) = 1C(i,i)=1.
3. Fill in the rest of the table using the recursive relation:
   * dp[i][j] = dp[i-1][j-1] + dp[i-1][j] for 0<j<i0 < j < i0<j<i.
4. The value of C(n,k)C(n, k)C(n,k) will be in dp[n][k].

Illustration :



Code : def binomial\_coefficient(n, k):

    # Initialize a 2D array with (n+1) x (k+1) dimensions

    dp = [[0 for \_ in range(k + 1)] for \_ in range(n + 1)]

    # Fill the table according to the algorithm

    for i in range(n + 1):

        for j in range(min(i, k) + 1):

            # Base cases

            if j == 0 or j == i:

                dp[i][j] = 1

            # Calculate value using previously computed values

            else:

                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]

    return dp[n][k]

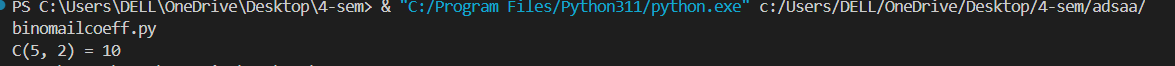
# Example usage

n = 5

k = 2

print(f"C({n}, {k}) = {binomial\_coefficient(n, k)}")  # Output: C(5, 2) = 10

Output :



1. Bellman Ford Algorithm :

**Date: 22/04/2024**

Aim :

the aim of the Bellman-Ford algorithm is to find the shortest path from a single source vertex to all other vertices in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative weight edges and can detect negative weight cycles.

Algorithm :

 Initialization:

* Set the distance to the source vertex to 0 and the distance to all other vertices to infinity.
* Create an array dist where dist[v] will hold the shortest distance from the source to vertex v.

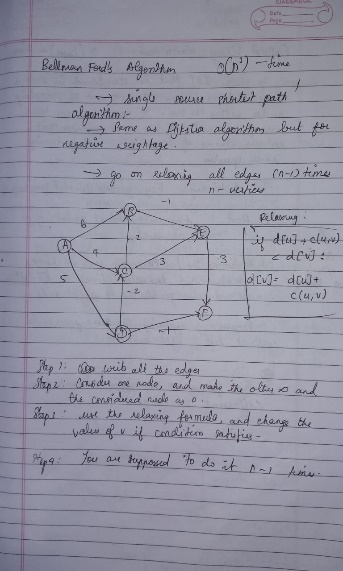
 Relaxation:

* For each vertex, apply relaxation for all edges. Repeat this process V-1 times where V is the number of vertices.
* Relaxation of an edge (u, v) with weight w involves checking if dist[u] + w < dist[v] and updating dist[v] if true.

 Negative Weight Cycle Check:

* After the V-1 iterations, perform one more iteration to check for negative weight cycles. If a shorter path is still found, then a negative weight cycle exists in the graph.

Illustration :



Code :

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.edges = []

    def add\_edge(self, u, v, w):

        self.edges.append((u, v, w))

    def bellman\_ford(self, source):

        # Step 1: Initialize distances from source to all other vertices as INFINITE

        dist = [float("inf")] \* self.V

        dist[source] = 0

        # Step 2: Relax all edges |V| - 1 times

        for \_ in range(self.V - 1):

            for u, v, w in self.edges:

                if dist[u] != float("inf") and dist[u] + w < dist[v]:

                    dist[v] = dist[u] + w

        # Step 3: Check for negative-weight cycles

        for u, v, w in self.edges:

            if dist[u] != float("inf") and dist[u] + w < dist[v]:

                print("Graph contains negative weight cycle")

                return None

        # No negative weight cycle found! Return distances

        return dist

# Example usage:

g = Graph(5)

g.add\_edge(0, 1, -1)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 3)

g.add\_edge(1, 3, 2)

g.add\_edge(1, 4, 2)

g.add\_edge(3, 2, 5)

g.add\_edge(3, 1, 1)

g.add\_edge(4, 3, -3)

source = 0

distances = g.bellman\_ford(source)

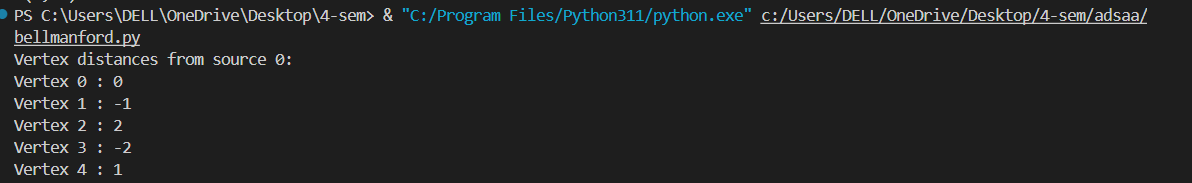
if distances:

    print(f"Vertex distances from source {source}:")

    for i in range(len(distances)):

        print(f"Vertex {i} : {distances[i]}")

Output:



Result :

The implementation of binomial co-efficient and BellmanFord algorithm using dynamic programming is successful and executed properly.

**Ex.No: 08 Implementation of Backtracking algorithms: Binomial**

**Date: 06/05/2024 N-queens,Hamilton circuit problem**

1. N-queens problem

Aim :

The aim of the N-Queens problem is to place N queens on an N×N chessboard such that no two queens threaten each other. This means that no two queens can be in the same row, column, or diagonal.

Algorithm :

 Initialization:

* Create a 2D array (or list of lists) to represent the N×N chessboard.

 Backtracking Function:

* Define a recursive function solve\_n\_queens(row) that attempts to place a queen in each column of the given row and then recursively attempts to place queens in subsequent rows.

 Check Validity:

* Before placing a queen in a column, check if it is safe to do so by ensuring that no other queen is already placed in the same column, or in the same diagonal (both left and right diagonals).

 Place the Queen:

* If it is safe to place the queen, mark the cell as occupied and recursively attempt to place queens in the next row.

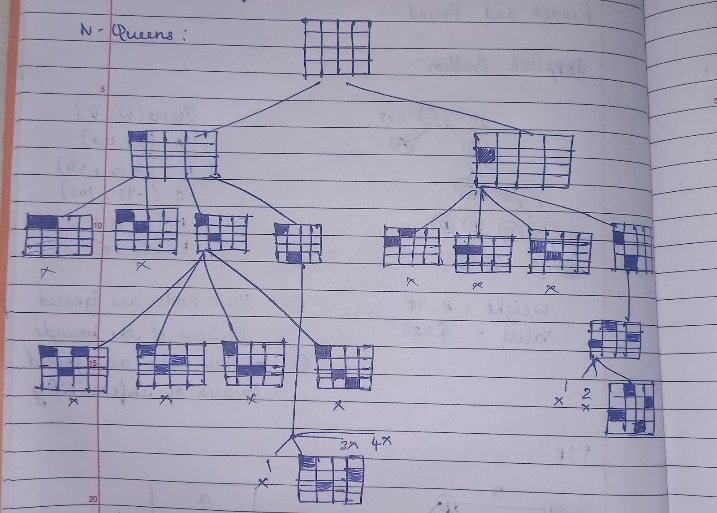
 Backtrack:

* If placing a queen in the current row and column leads to a conflict in subsequent rows, remove the queen (backtrack) and try the next column.

 Solution Found:

* If queens are placed successfully in all rows, record the solution.

Illustration :



Code :

def print\_solution(board):

    for row in board:

        print(" ".join(row))

    print()

def is\_safe(board, row, col):

    N = len(board)

    # Check column

    for i in range(row):

        if board[i][col] == 'Q':

            return False

    # Check left diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 'Q':

            return False

    # Check right diagonal

    for i, j in zip(range(row, -1, -1), range(col, N)):

        if board[i][j] == 'Q':

            return False

    return True

def solve\_n\_queens\_util(board, row):

    N = len(board)

    if row >= N:

        return True

    for col in range(N):

        if is\_safe(board, row, col):

            board[row][col] = 'Q'

            if solve\_n\_queens\_util(board, row + 1):

                return True

            board[row][col] = '.'

    return False

def solve\_n\_queens(N):

    board = [['.' for \_ in range(N)] for \_ in range(N)]

    if solve\_n\_queens\_util(board, 0):

        print\_solution(board)

    else:

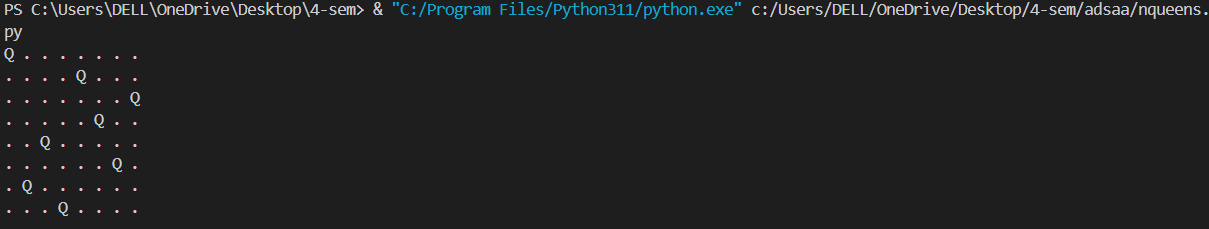
        print("No solution exists")

# Example usage

N = 8

solve\_n\_queens(N)

Output:



1. Hamilton circuit problem

Aim

The aim of the Hamiltonian Circuit problem is to determine if there exists a Hamiltonian circuit in a given graph. A Hamiltonian circuit is a closed loop on a graph where every vertex is visited exactly once, except for the starting vertex which is visited again at the end.

Algorithm :

 Initialization:

* Create an array path to store the vertices of the Hamiltonian circuit.
* Initialize the path with the starting vertex.

 Backtracking Function:

* Define a recursive function hamiltonian\_circuit(position) that tries to add vertices to the path starting from the current position.

 Check Validity:

* Check if the current vertex can be added to the Hamiltonian circuit by ensuring:
  + It is adjacent to the previously added vertex.
  + It has not been visited before.

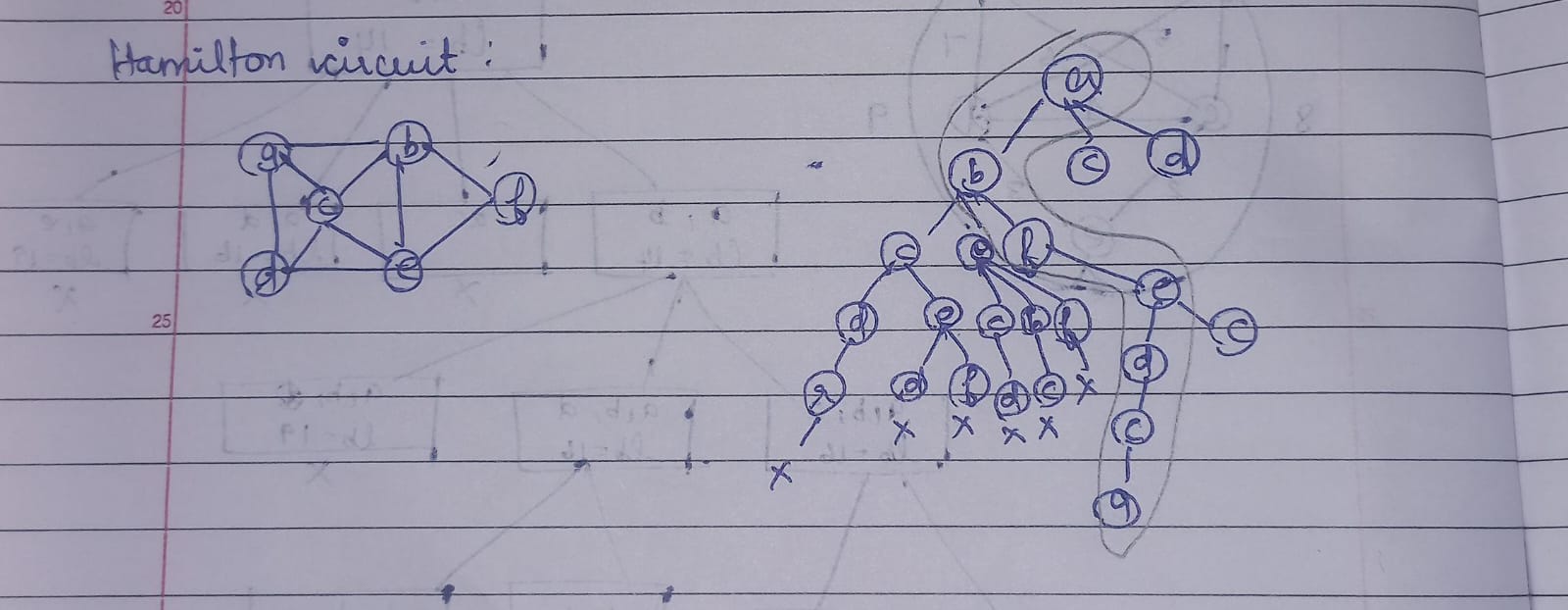
 Recursive Addition:

* If a valid vertex is found, add it to the path and recursively try to add the next vertex.
* If the path is complete and the last vertex is adjacent to the starting vertex, return true indicating a Hamiltonian circuit is found.

 Backtrack:

* If adding the current vertex does not lead to a solution, remove it (backtrack) and try the next vertex.

Illustration :



Code :

class HamiltonianCircuit:

    def \_\_init\_\_(self, graph):

        self.graph = graph

        self.V = len(graph)

        self.path = [-1] \* self.V

    def is\_safe(self, v, pos):

        # Check if this vertex is an adjacent vertex of the previously added vertex.

        if self.graph[self.path[pos - 1]][v] == 0:

            return False

        # Check if the vertex has already been included in the path.

        if v in self.path:

            return False

        return True

    def hamiltonian\_circuit\_util(self, pos):

        # Base case: If all vertices are included in the path

        if pos == self.V:

            # And if there is an edge from the last included vertex to the first vertex

            if self.graph[self.path[pos - 1]][self.path[0]] == 1:

                return True

            else:

                return False

        # Try different vertices as the next candidate in the Hamiltonian Circuit.

        for v in range(1, self.V):

            if self.is\_safe(v, pos):

                self.path[pos] = v

                if self.hamiltonian\_circuit\_util(pos + 1):

                    return True

                # Remove current vertex if it doesn't lead to a solution

                self.path[pos] = -1

        return False

    def find\_hamiltonian\_circuit(self):

        self.path[0] = 0  # Start from the first vertex

        if not self.hamiltonian\_circuit\_util(1):

            print("No Hamiltonian circuit found")

            return False

        self.print\_solution()

        return True

    def print\_solution(self):

        print("Hamiltonian Circuit exists:")

        for vertex in self.path:

            print(vertex, end=" ")

        print(self.path[0], "\n")  # Returning to the start vertex

# Example usage:

graph = [

    [0, 1, 0, 1, 0],

    [1, 0, 1, 1, 1],

    [0, 1, 0, 0, 1],

    [1, 1, 0, 0, 1],

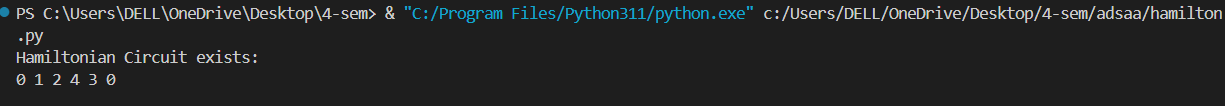
    [0, 1, 1, 1, 0]

]

hamiltonian = HamiltonianCircuit(graph)

hamiltonian.find\_hamiltonian\_circuit()

Output:



Result :

The N-queens problem and Hamilton circuit problem using backtracking were successfully implemented and executed.

**Ex.No: 09 Implementation of Iterative Improvement:**

**Date: 13/05/2024 Stable marriage and maximum flow problem**

1. Stable Marriage Problem

Aim:

The aim of the Stable Marriage Problem is to find a stable matching between two sets of elements, traditionally men and women, such that there are no two elements from different sets who would prefer each other over their current partners. A matching is stable if there are no two people who would rather be with each other than with their current partners.

Algorithm:

 Initialization:

* Each man and woman ranks all members of the opposite sex in order of preference.
* Each man is initially free and makes proposals to women based on his preference list.

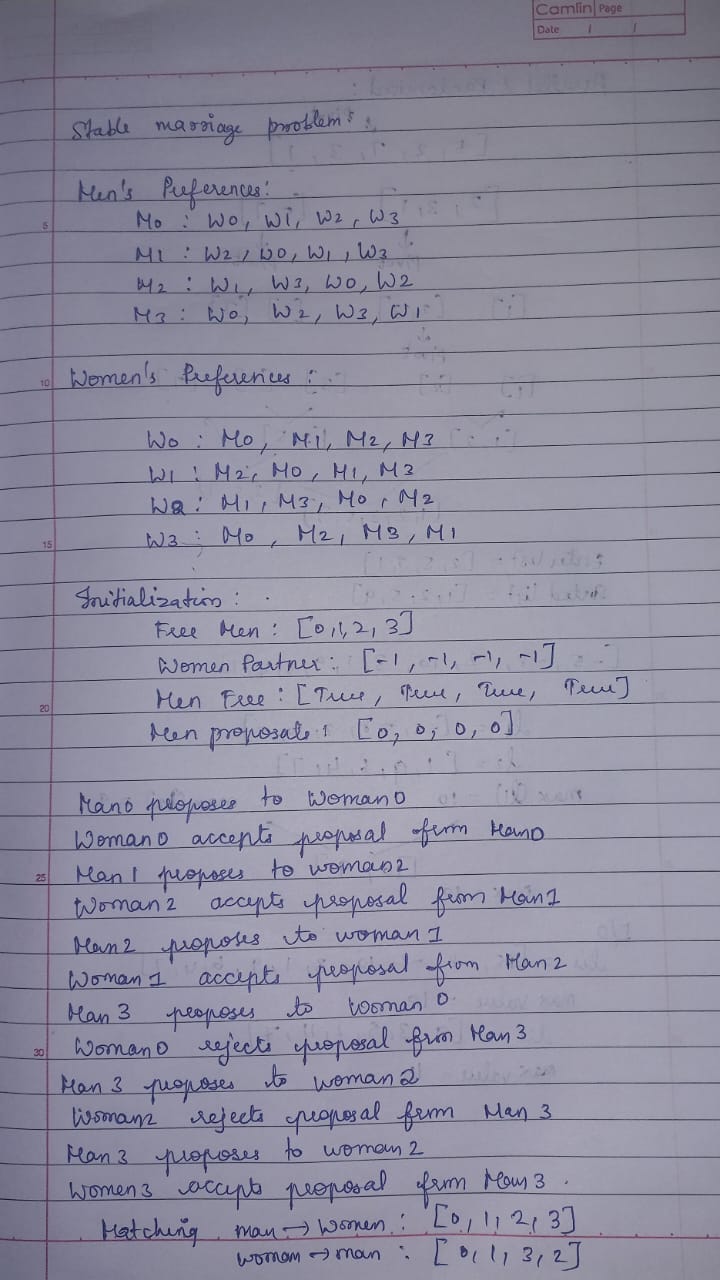
 Proposal Phase:

* Each free man proposes to the highest-ranked woman on his preference list who he has not yet proposed to.
* Each woman replies with "maybe" (provisionally accepting the proposal) if she is free or prefers this new proposal over her current engagement, and "no" otherwise.
* If the woman provisionally accepts the proposal, she becomes engaged to the man (if she was already engaged, the previous engagement is broken).

 Iterative Improvement:

* Repeat the proposal phase until no more men are free or no more proposals can be made.
* The algorithm ensures that no two people would rather be with each other than their current partners, leading to a stable matching.

Illustration :



Code :

def stable\_marriage(n, men\_preferences, women\_preferences):

    # Initialize all men and women as free

    free\_men = list(range(n))

    # None of the women are initially engaged

    women\_partner = [-1] \* n

    # Men's free status

    men\_free = [True] \* n

    # Number of women partners each man has proposed to

    men\_proposals = [0] \* n

    while free\_men:

        man = free\_men[0]

        # Find the first woman on this man's preference list to whom he has not yet proposed

        woman = men\_preferences[man][men\_proposals[man]]

        men\_proposals[man] += 1

        # If the woman is free, engage them

        if women\_partner[woman] == -1:

            women\_partner[woman] = man

            men\_free[man] = False

            free\_men.pop(0)

        else:

            # If the woman is not free, check if she prefers this man over her current partner

            current\_partner = women\_partner[woman]

            if women\_preferences[woman].index(man) < women\_preferences[woman].index(current\_partner):

                women\_partner[woman] = man

                men\_free[man] = False

                men\_free[current\_partner] = True

                free\_men.pop(0)

                free\_men.append(current\_partner)

            else:

                # If the woman prefers her current partner, the man remains free

                continue

    # Convert the partner list to the form of man -> woman

    men\_partner = [-1] \* n

    for woman in range(n):

        man = women\_partner[woman]

        men\_partner[man] = woman

    return men\_partner, women\_partner

# Example usage:

n = 4

men\_preferences = [

    [0, 1, 2, 3],

    [2, 0, 1, 3],

    [1, 3, 0, 2],

    [0, 2, 3, 1]

]

women\_preferences = [

    [0, 1, 2, 3],

    [2, 0, 1, 3],

    [1, 3, 0, 2],

    [0, 2, 3, 1]

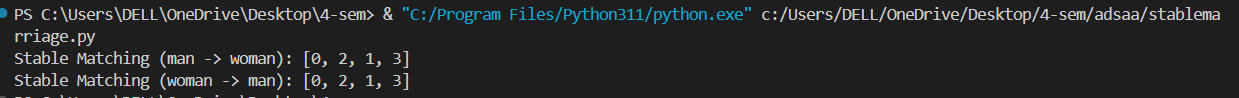
]

men\_matching, women\_matching = stable\_marriage(n, men\_preferences, women\_preferences)

print("Stable Matching (man -> woman):", men\_matching)

print("Stable Matching (woman -> man):", women\_matching)

Output:



Algorithm Analysis:

Time complexity:

* **Worst-case time complexity**: O(n^2) where n is the number of participants.
  + Each man proposes to at most n women, leading to O(n) proposals.
  + Since there are n men, this results in O(n^2) total operations.

1. Maximum Flow Problem

Aim:

The aim of the Maximum Flow Problem is to find the maximum flow that can be sent from a source node to a sink node in a flow network, subject to capacity constraints on the edges.

Algorithm: Ford-Fulkerson Algorithm with Edmonds-Karp Implementation

The Ford-Fulkerson algorithm is commonly used to solve the maximum flow problem. Here, we'll use the Edmonds-Karp implementation which uses Breadth-First Search (BFS) to find augmenting paths efficiently.

Algorithm :

 Initialization:

* Initialize flow on all edges to 0.
* Initialize residual capacities as the original capacities of edges.

 Find Augmenting Paths:

* Use BFS to find an augmenting path from the source to the sink in the residual graph.

 Update Flow:

* Update flow along the augmenting path found.
* Update the residual capacities of forward and backward edges.

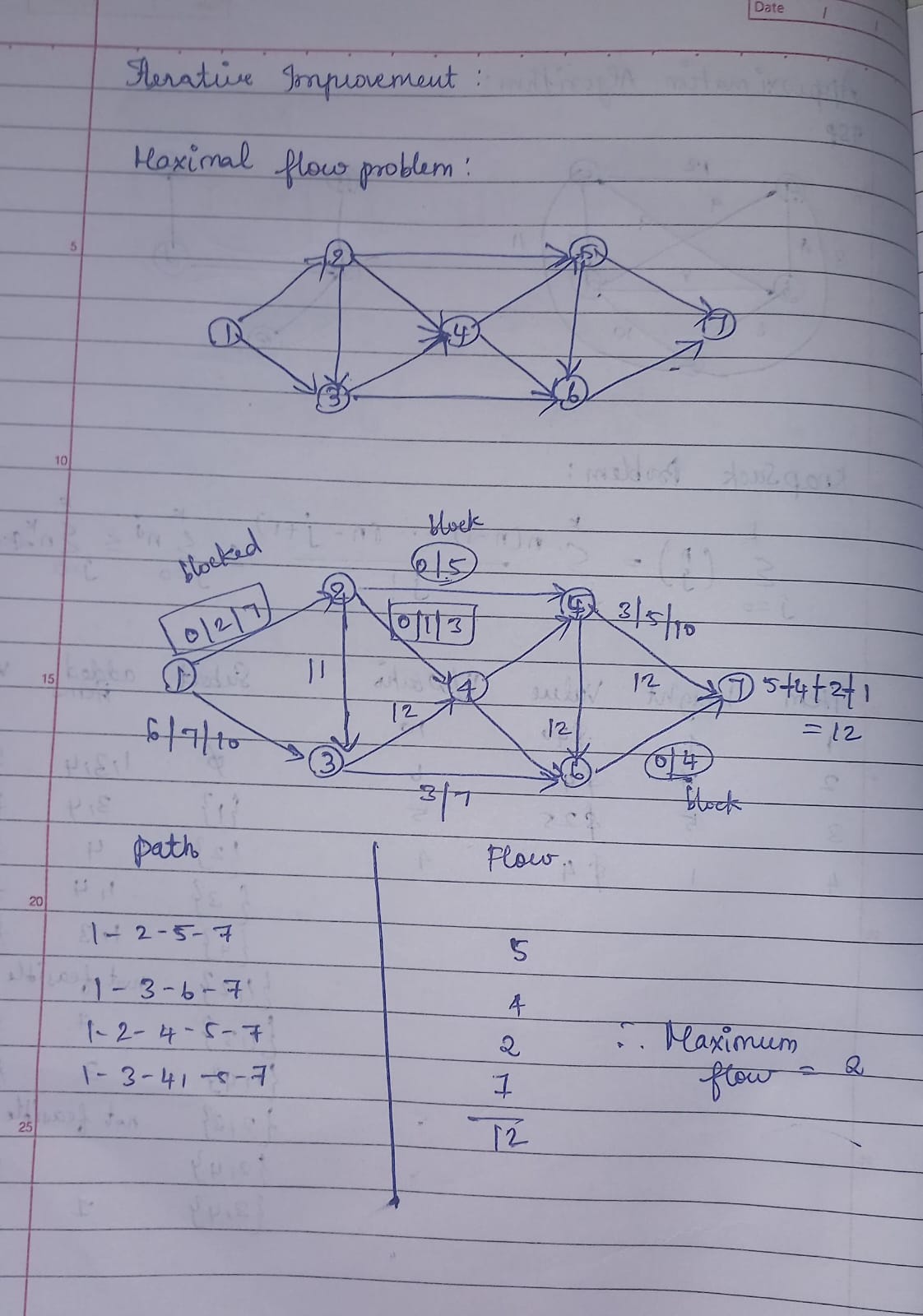
 Repeat:

* Repeat steps 2 and 3 until no augmenting paths can be found.

 Output:

* The maximum flow is the sum of flows into the sink.

Illustration :



Code :

from collections import defaultdict, deque

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(dict)

    def add\_edge(self, u, v, capacity):

        self.graph[u][v] = capacity

        self.graph[v][u] = 0  # Initialize backward edge capacity to 0

    def bfs(self, s, t, parent):

        visited = set()

        queue = deque()

        queue.append(s)

        visited.add(s)

        while queue:

            u = queue.popleft()

            for v, capacity in self.graph[u].items():

                if v not in visited and capacity > 0:

                    queue.append(v)

                    visited.add(v)

                    parent[v] = u

                    if v == t:

                        return True

        return False

    def ford\_fulkerson(self, source, sink):

        parent = {}

        max\_flow = 0

        while self.bfs(source, sink, parent):

            path\_flow = float('inf')

            s = sink

            while s != source:

                path\_flow = min(path\_flow, self.graph[parent[s]][s])

                s = parent[s]

            max\_flow += path\_flow

            v = sink

            while v != source:

                u = parent[v]

                self.graph[u][v] -= path\_flow

                self.graph[v][u] += path\_flow  # Update backward edge

                v = parent[v]

        return max\_flow

# Example usage:

g = Graph()

g.add\_edge('S', 'A', 10)

g.add\_edge('S', 'B', 5)

g.add\_edge('A', 'C', 15)

g.add\_edge('A', 'B', 15)

g.add\_edge('B', 'D', 10)

g.add\_edge('C', 'T', 10)

g.add\_edge('D', 'C', 5)

g.add\_edge('D', 'T', 10)

source = 'S'

sink = 'T'

max\_flow = g.ford\_fulkerson(source, sink)

print("Maximum Flow:", max\_flow)

Output:



Algorithm analysis:

* **Correctness**: Ford-Fulkerson algorithm finds the maximum flow by repeatedly finding augmenting paths until no more paths exist.
* **Termination**: The algorithm terminates when no augmenting path can be found.
* **Time Complexity**: O(V⋅E^2) where V is the number of vertices and E is the number of edges. The bottleneck is the BFS in each iteration which takes O€time, and there can be O(E) iterations in the worst case.

**Ex.No: 10 Implementation of Branch and Bound Technique:**

**Date: 20/05/2024 Knapsack Problem and TSP problem**

1. Knapsack Problem

Aim

The aim of the Knapsack Problem is to select a combination of items with maximum value while keeping the total weight within a given capacity.

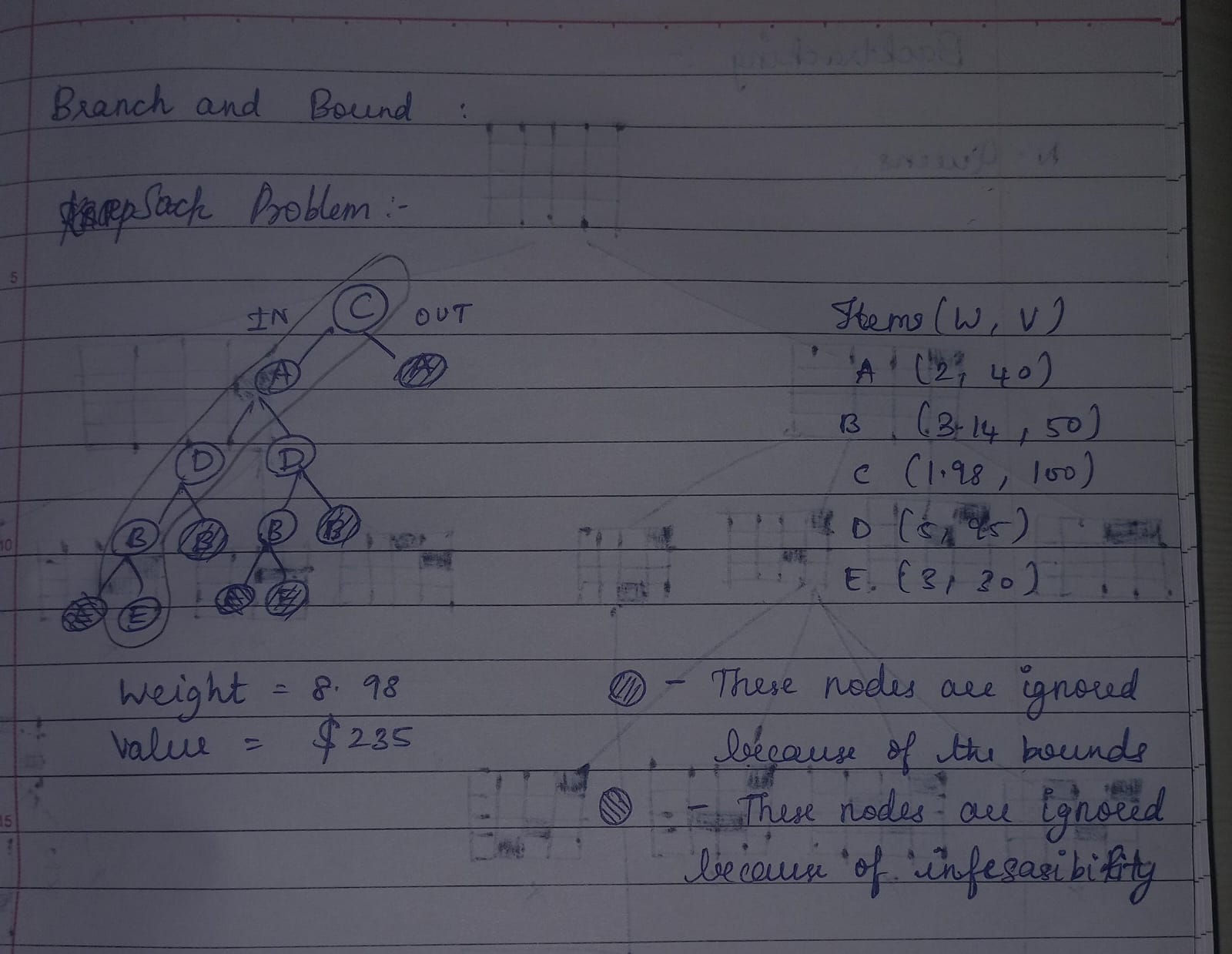
Algorithm: Branch and Bound Technique

The Branch and Bound technique is used to solve combinatorial optimization problems like the Knapsack Problem by systematically searching the solution space and pruning branches that cannot lead to an optimal solution.

Algorithm Steps:

1. Initialization:
   * Initialize the current best solution with zero value.
   * Start with an empty subset and compute its value and weight.
2. Branching:
   * Choose an item to include or exclude from the knapsack.
   * Create two branches: one including the item and one excluding the item.
3. Bounding:
   * Calculate the upper bound (maximum possible value) for each branch.
   * Prune the branch if its upper bound is less than the current best solution.
4. Recursion:
   * Recursively explore the branches until reaching the leaf nodes.
   * Update the current best solution if a better solution is found.
5. Termination:
   * Stop when all branches are explored.

Illustration :



Code :

class Item:

    def \_\_init\_\_(self, weight, value):

        self.weight = weight

        self.value = value

def knapsack\_branch\_bound(items, capacity):

    n = len(items)

    max\_value = 0

    best\_subset = [0] \* n

    def bound(current\_weight, current\_value, level):

        if current\_weight > capacity:

            return -1  # Infeasible solution

        bound\_value = current\_value

        j = level + 1

        total\_weight = current\_weight

        while j < n and total\_weight + items[j].weight <= capacity:

            total\_weight += items[j].weight

            bound\_value += items[j].value

            j += 1

        if j < n:

            bound\_value += (capacity - total\_weight) \* items[j].value / items[j].weight

        return bound\_value

    def knapsack\_recursive(level, current\_weight, current\_value):

        nonlocal max\_value, best\_subset

        if current\_weight <= capacity and current\_value > max\_value:

            max\_value = current\_value

            best\_subset = [1 if i in selected\_items else 0 for i in range(n)]

        if level == n:

            return

        if bound(current\_weight, current\_value, level) <= max\_value:

            return  # Prune the branch if its upper bound is less than max\_value

        selected\_items.add(level)

        knapsack\_recursive(level + 1, current\_weight + items[level].weight, current\_value + items[level].value)

        selected\_items.remove(level)

        knapsack\_recursive(level + 1, current\_weight, current\_value)

    selected\_items = set()

    knapsack\_recursive(0, 0, 0)

    return max\_value, best\_subset

# Example usage:

items = [Item(2, 10), Item(3, 20), Item(5, 30)]

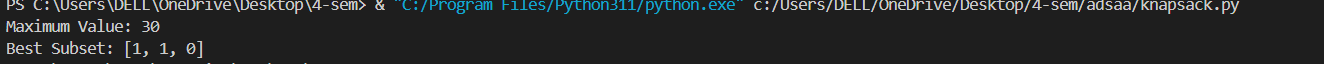
capacity = 8

max\_value, best\_subset = knapsack\_branch\_bound(items, capacity)

print("Maximum Value:", max\_value)

print("Best Subset:", best\_subset)

Output:



Algorithm Analysis:

* Why Branch and Bound?
  + Branch and Bound technique is used to solve optimization problems by exploring the solution space efficiently.
  + It helps in pruning the branches of the search tree that cannot lead to an optimal solution, thus reducing the search space.
* Time Complexity:
  + The time complexity depends on the number of nodes in the search tree.
  + In the worst case, all nodes of the tree need to be explored, leading to exponential time complexity.
  + However, pruning techniques reduce the search space significantly, leading to better performance in practice.

1. TSP problem

Aim

The aim of the Traveling Salesman Problem is to find the shortest possible route that visits each city exactly once and returns to the original city.

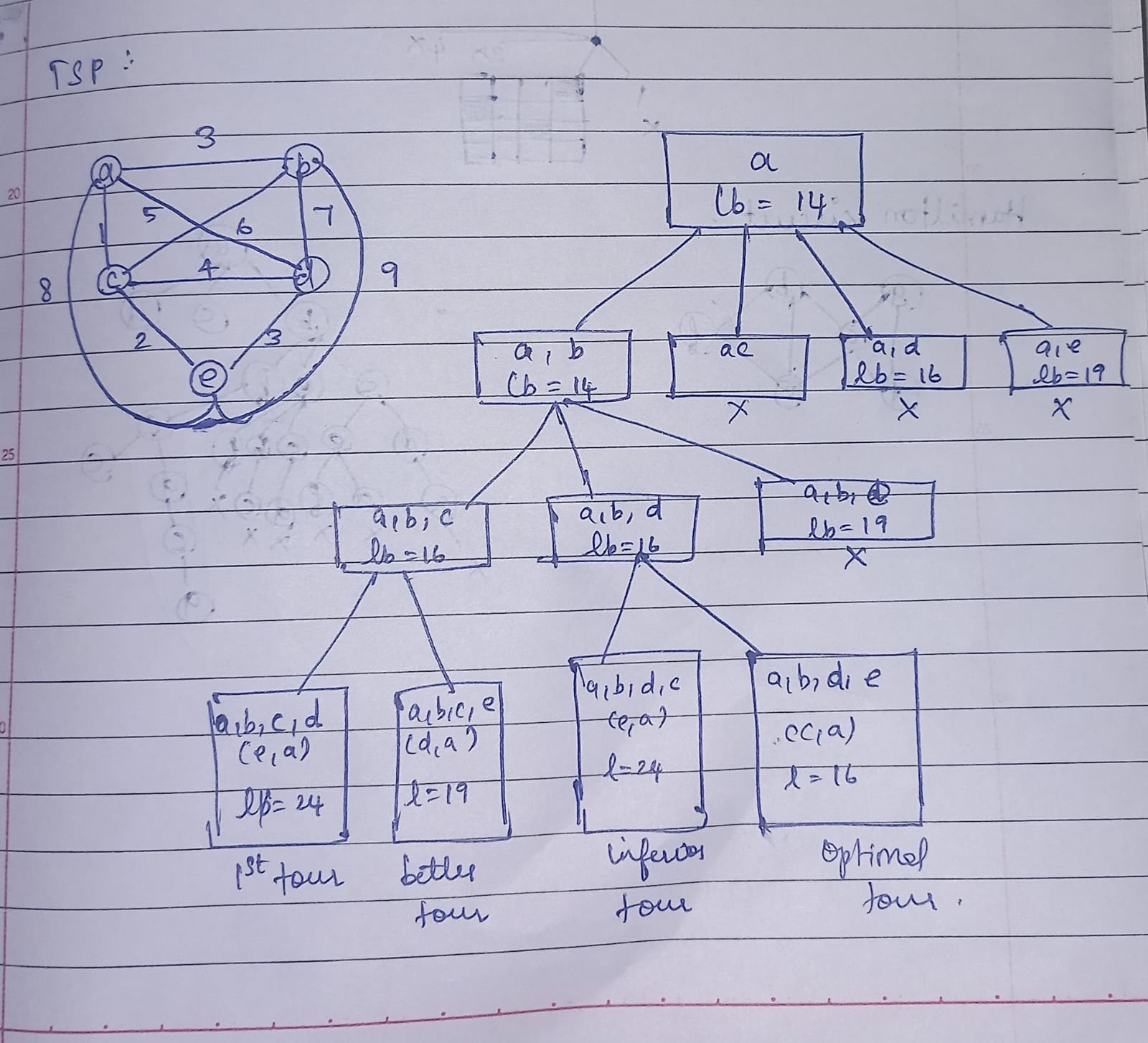
Algorithm: Branch and Bound Technique

The Branch and Bound technique is used to solve optimization problems like the TSP by systematically exploring the solution space and pruning branches that cannot lead to an optimal solution.

Algorithm Steps:

1. Initialization:
   * Initialize the current best solution with infinity.
   * Start with the first city as the current city and an empty tour.
2. Branching:
   * Choose a city to visit next.
   * Create branches for each unvisited city.
3. Bounding:
   * Calculate the lower bound (minimum possible distance) for each branch.
   * Prune the branch if its lower bound is greater than the current best solution.
4. Recursion:
   * Recursively explore the branches until all cities are visited.
   * Update the current best solution if a better solution is found.
5. Termination:
   * Stop when all branches are explored.

Illustration:



Code :

import sys

class TSP:

    def \_\_init\_\_(self, graph):

        self.graph = graph

        self.n = len(graph)

        self.visited = [False] \* self.n

        self.min\_cost = sys.maxsize

        self.best\_path = []

    def tsp\_branch\_bound(self):

        def bound(path):

            cost = 0

            for i in range(self.n):

                if not self.visited[i]:

                    min\_cost = min(self.graph[path[-1]][i] for i in range(self.n) if i != path[-1])

                    cost += min\_cost

            return cost

        def tsp\_recursive(path, cost):

            if len(path) == self.n:

                cost += self.graph[path[-1]][path[0]]

                if cost < self.min\_cost:

                    self.min\_cost = cost

                    self.best\_path = path[:]

                return

            for i in range(self.n):

                if not self.visited[i]:

                    new\_cost = cost + self.graph[path[-1]][i]

                    if new\_cost + bound(path) < self.min\_cost:

                        path.append(i)

                        self.visited[i] = True

                        tsp\_recursive(path, new\_cost)

                        path.pop()

                        self.visited[i] = False

        start\_city = 0  # Starting from city 0

        tsp\_recursive([start\_city], 0)

        return self.min\_cost, self.best\_path

# Example usage:

graph = [

    [0, 10, 15, 20],

    [10, 0, 35, 25],

    [15, 35, 0, 30],

    [20, 25, 30, 0]

]

tsp\_solver = TSP(graph)

min\_cost, best\_path = tsp\_solver.tsp\_branch\_bound()

print("Minimum Cost:", min\_cost)

print("Best Path:", best\_path)

Output: 

Algorithm Analysis:

* Why Branch and Bound?
  + Branch and Bound is used to efficiently explore the solution space and find the optimal solution for the TSP.
  + It helps in pruning the branches of the search tree that cannot lead to an optimal solution, thus reducing the search space.
* Time Complexity:
  + The time complexity depends on the number of nodes in the search tree.
  + In the worst case, all nodes of the tree need to be explored, leading to exponential time complexity.
  + However, pruning techniques reduce the search space significantly, leading to better performance in practice.

**Ex.No: 11 Implementation of Approximation Algorithm:**

**Date: 27/05/2024 Knapsack Problem and TSP problem**

1. Knapsack Problem

Aim

The aim of the approximation algorithm for the Knapsack Problem is to find a solution that is close to the optimal solution but with less computational complexity.

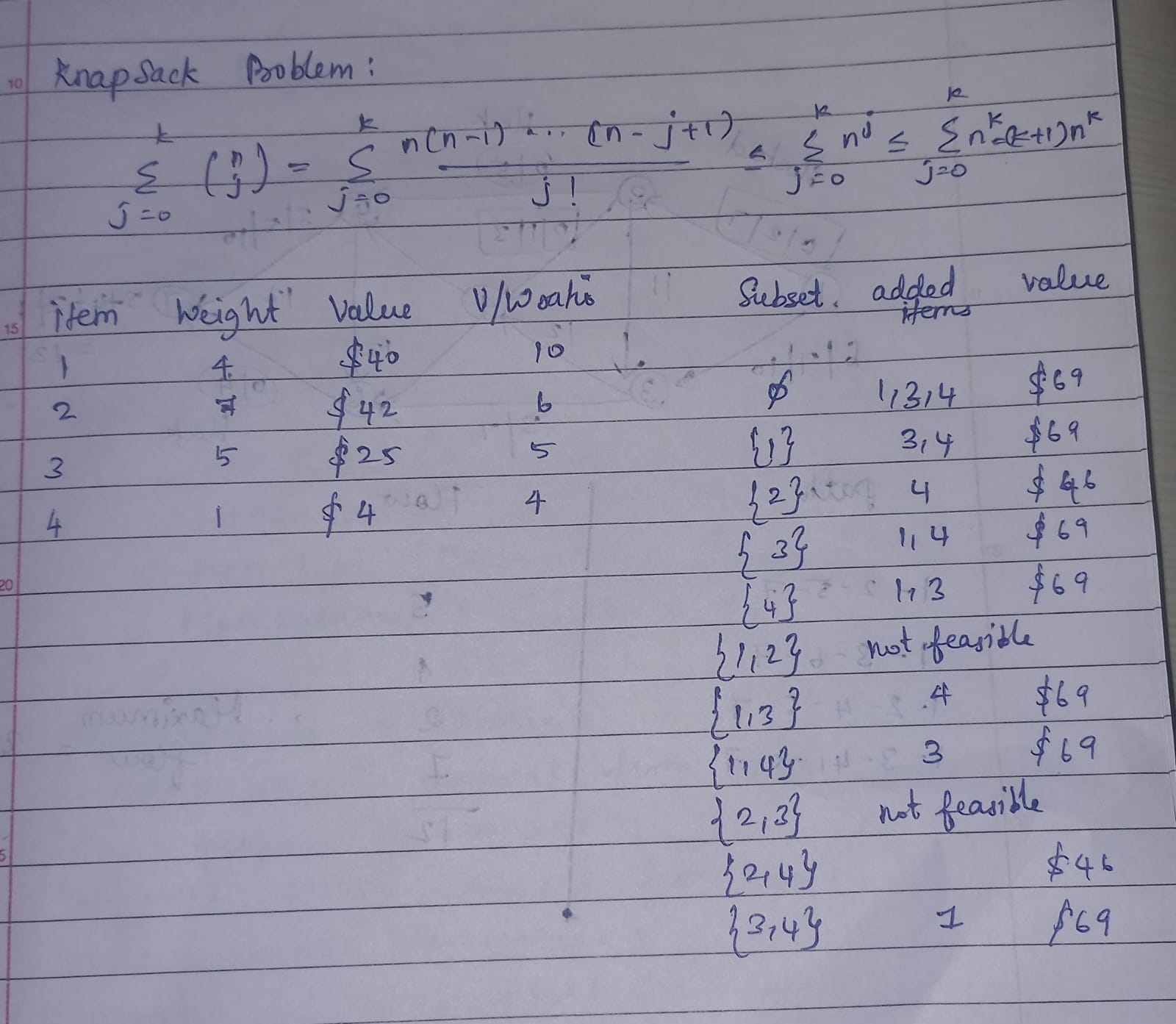
Algorithm

The approximation algorithm for the Knapsack Problem is based on a greedy approach where items are selected based on their value-to-weight ratio.

Algorithm Steps:

1. Sort Items:
   * Sort the items based on their value-to-weight ratio in non-increasing order.
2. Select Items:
   * Start with an empty knapsack.
   * Iterate through the sorted items and add them to the knapsack if they fit.
3. Greedy Selection:
   * At each step, select the item with the highest value-to-weight ratio that can fit into the knapsack.

Illustration :



Code :

class Item:

    def \_\_init\_\_(self, weight, value):

        self.weight = weight

        self.value = value

def knapsack\_approximation(items, capacity):

    # Sort items based on value-to-weight ratio in non-increasing order

    sorted\_items = sorted(items, key=lambda x: x.value / x.weight, reverse=True)

    knapsack = []

    total\_weight = 0

    total\_value = 0

    for item in sorted\_items:

        if total\_weight + item.weight <= capacity:

            knapsack.append(item)

            total\_weight += item.weight

            total\_value += item.value

    return total\_value, knapsack

# Example usage:

items = [Item(2, 10), Item(3, 20), Item(5, 30)]

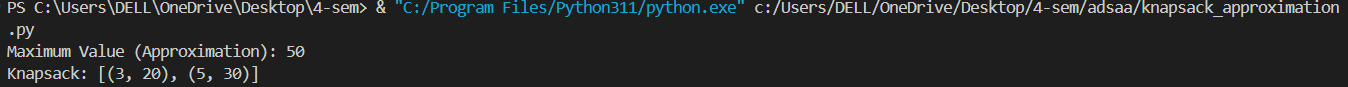
capacity = 8

max\_value, knapsack = knapsack\_approximation(items, capacity)

print("Maximum Value (Approximation):", max\_value)

print("Knapsack:", [(item.weight, item.value) for item in knapsack])

Output:



Algorithm Analysis:

* Time Complexity:
  + Sorting the items based on the value-to-weight ratio takes O(nlogn) time.
  + Greedy selection of items takes O(n) time.
  + Overall time complexity O(nlogn).
* Approximation Factor:
  + The approximation factor of this algorithm depends on the sorting order.
  + For this greedy approach, the approximation factor is 1−1/, where e is Euler's number (approximately 2.718).
  + This means the solution is guaranteed to be within 63.2%of the optimal solution.
* Why This Technique is Used?
  + Approximation algorithms provide a fast way to find a near-optimal solution for NP-hard problems like the Knapsack Problem.
  + While the solution may not be optimal, it provides a good balance between solution quality and computational complexity.

1. TSP problem

Aim

The aim of the approximation algorithm for the Traveling Salesman Problem (TSP) is to find a solution that is close to the optimal tour length while having less computational complexity.

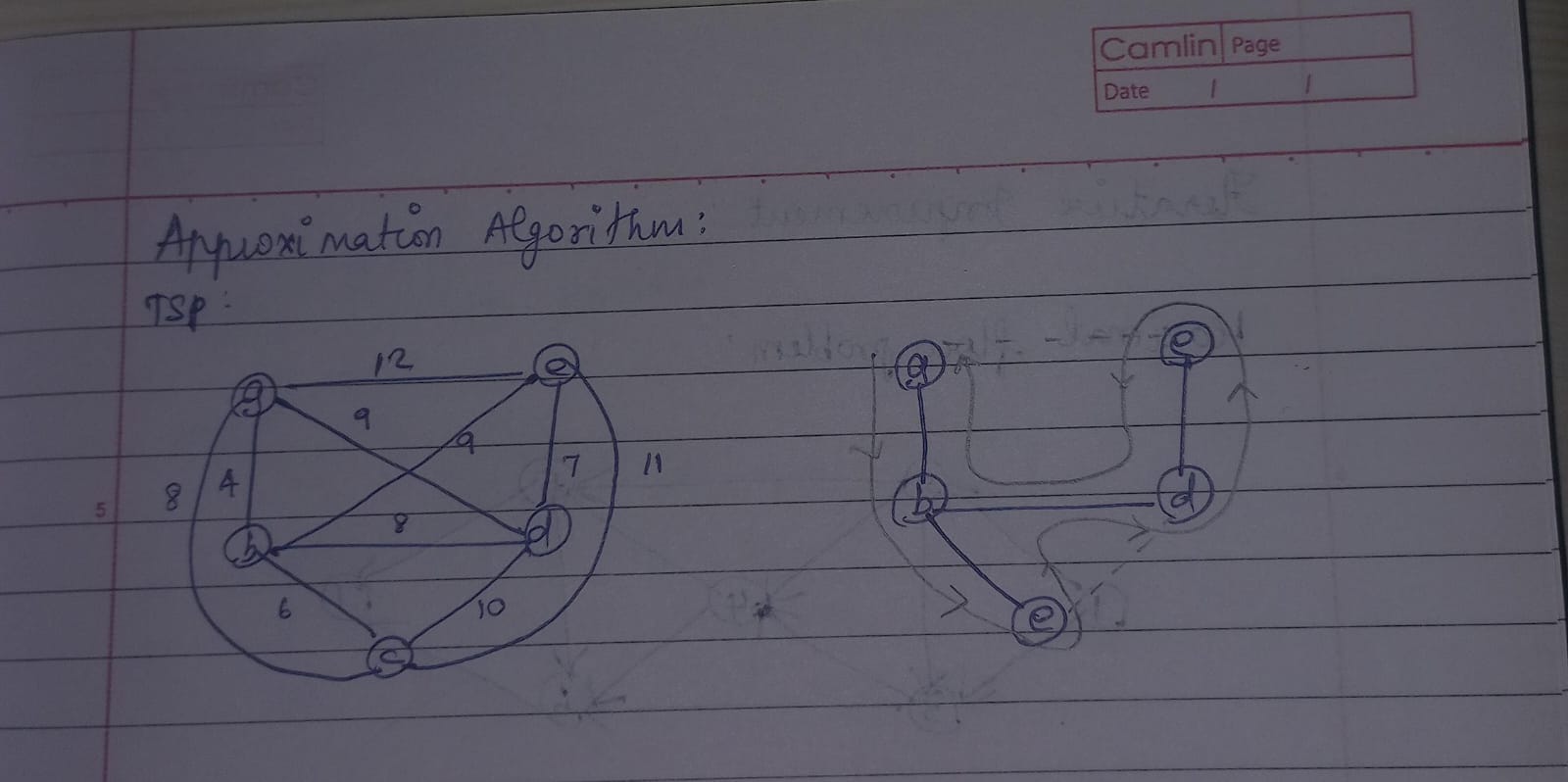
Algorithm

The approximation algorithm for TSP is based on the Minimum Spanning Tree (MST) approach, often known as the Minimum Spanning Tree heuristic.

Algorithm Steps:

1. Minimum Spanning Tree (MST):
   * Construct a Minimum Spanning Tree of the given graph.
2. Depth-First Traversal:
   * Perform a depth-first traversal of the MST to create a tour.
3. Tour Completion:
   * Complete the tour by returning to the starting node.

Illustration :



Code :

import networkx as nx

def tsp\_approximation(graph):

    # Step 1: Construct Minimum Spanning Tree

    mst = nx.minimum\_spanning\_tree(graph)

    # Step 2: Depth-First Traversal

    tour = list(nx.dfs\_preorder\_nodes(mst, source=0))

    # Step 3: Tour Completion

    tour.append(tour[0])  # Return to the starting node to complete the tour

    tour\_length = sum(graph[tour[i]][tour[i + 1]]['weight'] for i in range(len(tour) - 1))

    return tour, tour\_length

# Example usage:

graph = nx.Graph()

graph.add\_weighted\_edges\_from([(0, 1, 10), (0, 2, 15), (0, 3, 20),

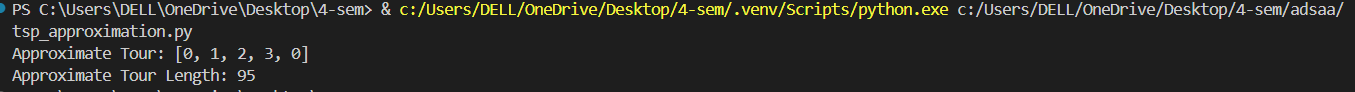
                               (1, 2, 35), (1, 3, 25), (2, 3, 30)])

tour, tour\_length = tsp\_approximation(graph)

print("Approximate Tour:", tour)

print("Approximate Tour Length:", tour\_length)

Output:



Algorithm Analysis:

* Time Complexity:
  + Constructing the Minimum Spanning Tree using Kruskal's or Prim's algorithm takes O(ElogV) time.
  + Depth-First Traversal takes O(V+E) time.
  + Overall time complexity: O(ElogV) or O(V+E), depending on the MST construction method.
* Approximation Factor:
  + The approximation factor of this algorithm is 2, meaning the tour length found by this algorithm is at most twice the length of the optimal tour.
* Why This Technique is Used?
  + Approximation algorithms provide a fast way to find a near-optimal solution for NP-hard problems like TSP.
  + While the solution may not be optimal, it provides a good balance between solution quality and computational complexity.