**Implementation of splay trees**

1. **Insertion and display in splay tree**

**Aim:**

To implement Splay Tree’s Insertion Operation using Data Structures and algorithms.

**Problem Description:**

A splay tree is a self-balancing binary search tree where the most recently accessed/inserted node becomes the root.

**Insertion Algorithm:**

* Start by performing a standard binary search tree insertion. Compare the value to be inserted with the root node of the tree. If the tree is empty, create a new node with the value and make it the root.
* If the value is smaller than the root, go to the left subtree and repeat step 1. If the value is larger than the root, go to the right subtree and repeat step 1.
* Once you find the appropriate position for the new value, create a new node and insert it as a leaf node in the appropriate subtree.
* After inserting the new node, perform a series of rotations to splay the newly inserted node to the root of the tree. This step helps optimize the tree structure by bringing frequently accessed nodes closer to the root.
* To perform the rotations, start by checking the relationship between the new node and its parent. There are three cases to consider:
* Zig-Zig Case: If both the new node and its parent are left children or both are right children, perform a double rotation (zig-zig rotation) to bring them to the root. This involves rotating the parent with its parent and then rotating the new node with its parent.
* Zig-Zag Case: If the new node is a left child and its parent is a right child (or vice versa), perform a single rotation (zig-zag rotation) to bring them to the root. This involves rotating the new node with its parent.
* Zig Case: If the new node is a left child and its parent is also a left child (or the new node is a right child and its parent is also a right child), perform a single rotation (zig rotation) to bring them to the root. This involves rotating the parent with its grandparent.
* Repeat step 5 until the new node becomes the root of the tree. This ensures that the new node is moved up the tree closer to the root, optimizing the tree structure for future operations.

**Code:**

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

        self.parent = None

class SplayTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, Newnode):

        if self.root == None:

          self.root = Newnode

        else:

            last = self.root

            while True:

                if Newnode.data < last.data:

                    if last.left == None:

                        last.left = Newnode

                        last.left.parent = last

                        self.splay(last.left)

                        break

                    else:

                        last = last.left

                else:

                    if last.right == None:

                        last.right = Newnode

                        last.right.parent = last

                        self.splay(last.right)

                        break

                    else:

                        last = last.right

    def left\_rotate(self, root):

        k = root.parent

        if k == self.root:

            m = root.left

            root.left = k

            k.parent = root

            k.right = m

            if m != None:

                m.parent = k

            self.root = root

        else:

            m = root.left

            root.left = k

            root.parent = k.parent

            k.parent = root

            k.right = m

            if m != None:

                m.parent = k

    def right\_rotate(self, root):

        k = root.parent

        if k == self.root:

            m = root.right

            root.right = k

            k.parent = root

            k.left = m

            if m != None:

                m.parent = k

            self.root = root

        else:

            m = root.right

            root.right = k

            root.parent = k.parent

            k.parent = root

            k.left = m

            if m != None:

                m.parent = k

    def splay(self, root):

        k = root.parent

        if k == self.root:

            if k.left == root:

                self.right\_rotate(root)

            else:

                self.left\_rotate(root)

        else:

            if root == k.left and k == k.parent.left:

                self.right\_rotate(k)

                self.right\_rotate(root)

            elif root == k.left and k == k.parent.right:

                self.right\_rotate(root)

                self.left\_rotate(root)

            elif root == k.right and k == k.parent.left:

                self.left\_rotate(root)

                self.right\_rotate(root)

            else:

                self.left\_rotate(k)

                self.left\_rotate(root)

        if root != self.root:

            self.splay(root)

    def inorder(self, root):

        if root:

            self.inorder(root.left)

            print(root.data)

            self.inorder(root.right)

    def find\_max(self, root):

        if root.right == None:

            return root

        else:

            self.find\_max(root.right)

splayTree = SplayTree()

while True:

n = int(input("Enter the no. of nodes:"))

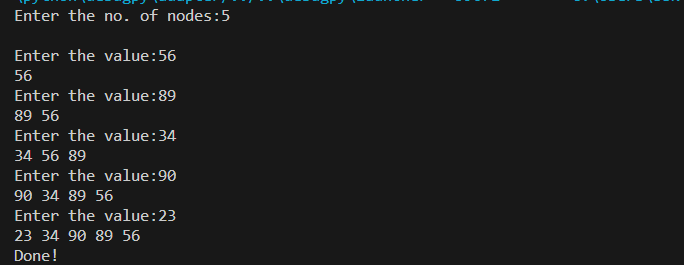
for i in range(n):

splayTree.insert(int(input("\nEnter the value:")))

splayTree.preorder(splayTree.root)

print("\nDone!")

**Output:**

****

**Time Complexity And Algorithm Analysis:**

1. **Time Complexity:**

• The time complexity of splay tree insertion is O(log n) in the average case and amortized O(log n) in the worst case, where n is the number of nodes in the tree.

• The amortized complexity takes into account the rebalancing operations performed during the insertion process.

1. **Algorithm Analysis:**

• The splay tree insertion algorithm consists of the following steps:

a) Perform a standard BST insertion to place the new node in the appropriate position based on its key value.

b) Splay the newly inserted node by applying a series of rotations to bring it to the root.

• The splaying process involves zig-zig, zig-zag, or zig operations to restructure the tree and maintain the self-adjusting property.

• The goal of splaying is to move the newly inserted node to the root to optimize future access times.

**Result:**

Thus the Splay tree’s Insertion Operation has been Implemented succesfully.

**b. Searching and deletion in splay tree**

**Aim:**

To implement searching and deletion in the Splay Tree data structure.

**Problem Description:**

A splay tree is a self-balancing binary search tree where the most recently accessed/inserted node becomes the root.

**Algorithm:**

**Searching Algorithm:**

1. Start at the root of the Splay Tree.
2. If the key of the current node matches the search key, perform a splay operation to bring the node to the root and exit.
3. If the search key is less than the current node's key, move to the left child.
4. If the search key is greater than the current node's key, move to the right child.
5. Repeat steps 2-4 until a match is found or the tree is exhausted.
6. If no match is found, perform a final splay operation on the last accessed node.

**Deletion Algorithm:**

1. Start at the root of the Splay Tree.
2. If the key of the node to be deleted matches the current node's key, perform a splay operation to bring the node to the root.
3. If the key of the node to be deleted is less than the current node's key, move to the left child. If the key of the node to be deleted is greater than the current node's key, move to the right child.
4. Repeat steps 2-4 until the node to be deleted is found or the tree is exhausted.
5. If the node to be deleted is not found, exit the algorithm.
6. Perform a splay operation on the node to be deleted, bringing it to the root.
7. If the node to be deleted has no left child, make its right child the new root.
8. If the node to be deleted has no right child, make its left child the new root.
9. If the node to be deleted has both left and right children, find the maximum node in its left subtree (or minimum node in the right subtree).
10. Perform a splay operation on the found node to bring it to the root.
11. Set its right child as the right child of the new root.
12. Set its left child as the left child of the new root.
13. Make the new root the root of the Splay Tree.

**Code:**

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

        self.parent = None

class Splay\_Tree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, Newnode):

        if self.root == None:

            self.root = Newnode

        else:

            last = self.root

            while True:

                if Newnode.data < last.data:

                    if last.left == None:

                        last.left = Newnode

                        last.left.parent = last

                        self.splay(last.left)

                        break

                    else:

                        last = last.left

                else:

                    if last.right == None:

                        last.right = Newnode

                        last.right.parent = last

                        self.splay(last.right)

                        break

                    else:

                        last = last.right

    def left\_rotate(self, root):

        k = root.parent

        if k == self.root:

            m = root.left

            root.left = k

            k.parent = root

            k.right = m

            if m != None:

                m.parent = k

            self.root = root

        else:

            m = root.left

            root.left = k

            root.parent = k.parent

            k.parent = root

            k.right = m

            if m != None:

                m.parent = k

    def right\_rotate(self, root):

        k = root.parent

        if k == self.root:

            m = root.right

            root.right = k

            k.parent = root

            k.left = m

            if m != None:

                m.parent = k

            self.root = root

        else:

            m = root.right

            root.right = k

            root.parent = k.parent

            k.parent = root

            k.left = m

            if m != None:

                m.parent = k

    def splay(self, root):

        k = root.parent

        if k == self.root:

            if k.left == root:

                self.right\_rotate(root)

            else:

                self.left\_rotate(root)

        else:

            if root == k.left and k == k.parent.left:

                self.right\_rotate(k)

                self.right\_rotate(root)

            elif root == k.left and k == k.parent.right:

                self.right\_rotate(root)

                self.left\_rotate(root)

            elif root == k.right and k == k.parent.left:

                self.left\_rotate(root)

                self.right\_rotate(root)

            else:

                self.left\_rotate(k)

                self.left\_rotate(root)

        if root != self.root:

            self.splay(root)

    def search(self, root, Node):

        if root:

            if root.data == Node.data:

                self.splay(root)

                print("Node found:", self.root.data)

            else:

                self.search(root.left, Node)

                self.search(root.right, Node)

    def inorder(self, root):

        if root:

            self.inorder(root.left)

            print(root.data)

            self.inorder(root.right)

    def find\_max(self, root):

        if root.right == None:

            return root

        else:

            self.find\_max(root.right)

    def delete(self, root, Node):

        if root:

            if root.data == Node.data:

                if root == self.root:

                    if root.left == None:

                        self.root = root.right

                    else:

                        self.splay(self.find\_max(root.left))

                        self.root.right = root.right

                else:

                    self.splay(root)

                    self.splay(self.find\_max(root.left))

                    self.root.right = root.right

            else:

                self.delete(root.left, Node)

                self.delete(root.right, Node)

splayTree = SplayTree()

while True:

    n = int(input("Enter the no. of nodes:"))

    for i in range(n):

            splayTree.insert(int(input("\nEnter the value:")))

            splayTree.preorder(splayTree.root)

    print("Done!")

    print("Serach Operation:")

    splayTree.search(int(input("\nEnter the value to search: ")))

    splayTree.preorder(splayTree.root)

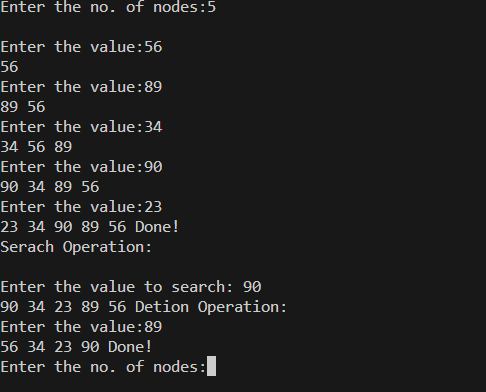
    print("Detion Operation:")

    splayTree.delete(int(input("Enter the value:")))

    splayTree.preorder(splayTree.root)

    print("Done!")

**Output:**

****

**Time Complexity And Algorithm Analysis:**

1. **Time Complexity:**

• The time complexity of splay tree insertion is O(log n) in the average case and amortized O(log n) in the worst case, where n is the number of nodes in the tree.

• The amortized complexity takes into account the rebalancing operations performed during the insertion process.

1. **Algorithm Analysis:**

• The splay tree insertion algorithm consists of the following steps:

a) Perform a standard BST insertion to place the new node in the appropriate position based on its key value.

b) Splay the newly inserted node by applying a series of rotations to bring it to the root.

• The splaying process involves zig-zig, zig-zag, or zig operations to restructure the tree and maintain the self-adjusting property.

• The goal of splaying is to move the newly inserted node to the root to optimize future access times.

**Result:**

Thus, insertion, deletion and searching in splay tree has been done successfully.

**Implementation of B+ trees**

**Aim:**

To implement the B+ tree data structure.

**Problem description:**

B+ tree is a data structure that optimizes searching by storing all elements at its leaf nodes.

**Algorithm:**

**Insertion:**

1. Start at the root of the tree and traverse down to a leaf node, following the same steps as in the search algorithm.
2. If the key is already present, update the associated value.
3. If the leaf node has space for the new key-value pair, insert it in the correct sorted position.
4. If the leaf node is full, split it into two and promote the middle key to the parent node.
5. If the parent node is full, recursively split it, promoting the middle key to its parent, and so on.
6. Adjust the pointers accordingly to maintain the tree structure.

**Searching:**

1. Start at the root of the tree.
2. Compare the search key with the keys stored in the current node.
3. If the key is found, return the associated value.
4. If the key is less than the current key, follow the left child pointer.
5. If the key is greater, follow the right child pointer.
6. Repeat the above steps until reaching a leaf node or finding the key.

**Deletion:**

1. Start at the root and traverse down to the leaf node containing the key to delete, following the same steps as in the search algorithm.
2. If the key is found, remove it from the leaf node.
3. If the node underflows (has fewer keys than the minimum allowed), perform the following operations:
4. Borrow a key from its adjacent sibling if the sibling has more than the minimum keys.
5. If borrowing is not possible, merge the node with its sibling, adjusting the pointers.
6. Propagate the underflow condition to the parent node, performing the same operations recursively if necessary.
7. Update the keys in the parent nodes accordingly, if necessary.

**Code:**

import math

class Node:

    def \_\_init\_\_(self, order):

        self.order = order

        self.values = []

        self.keys = []

        self.nextKey = None

        self.parent = None

        self.check\_leaf = False

    def insert\_at\_leaf(self, leaf, value, key):

        if (self.values):

            temp1 = self.values

            for i in range(len(temp1)):

                if (value == temp1[i]):

                    self.keys[i].append(key)

                    break

                elif (value < temp1[i]):

                    self.values = self.values[:i] + [value] + self.values[i:]

                    self.keys = self.keys[:i] + [[key]] + self.keys[i:]

                    break

                elif (i + 1 == len(temp1)):

                    self.values.append(value)

                    self.keys.append([key])

                    break

        else:

            self.values = [value]

            self.keys = [[key]]

class BplusTree:

    def \_\_init\_\_(self, order):

        self.root = Node(order)

        self.root.check\_leaf = True

    def insert(self, value, key):

        value = str(value)

        old\_node = self.search(value)

        old\_node.insert\_at\_leaf(old\_node, value, key)

        if (len(old\_node.values) == old\_node.order):

            node1 = Node(old\_node.order)

            node1.check\_leaf = True

            node1.parent = old\_node.parent

            mid = int(math.ceil(old\_node.order / 2)) - 1

            node1.values = old\_node.values[mid + 1:]

            node1.keys = old\_node.keys[mid + 1:]

            node1.nextKey = old\_node.nextKey

            old\_node.values = old\_node.values[:mid + 1]

            old\_node.keys = old\_node.keys[:mid + 1]

            old\_node.nextKey = node1

            self.insert\_in\_parent(old\_node, node1.values[0], node1)

    def search(self, value):

        current\_node = self.root

        while(current\_node.check\_leaf == False):

            temp2 = current\_node.values

            for i in range(len(temp2)):

                if (value == temp2[i]):

                    current\_node = current\_node.keys[i + 1]

                    break

                elif (value < temp2[i]):

                    current\_node = current\_node.keys[i]

                    break

                elif (i + 1 == len(current\_node.values)):

                    current\_node = current\_node.keys[i + 1]

                    break

        return current\_node

    def find(self, value, key):

        l = self.search(value)

        for i, item in enumerate(l.values):

            if item == value:

                if key in l.keys[i]:

                    return True

                else:

                    return False

        return False

    def insert\_in\_parent(self, n, value, ndash):

        if (self.root == n):

            rootNode = Node(n.order)

            rootNode.values = [value]

            rootNode.keys = [n, ndash]

            self.root = rootNode

            n.parent = rootNode

            ndash.parent = rootNode

            return

        parentNode = n.parent

        temp3 = parentNode.keys

        for i in range(len(temp3)):

            if (temp3[i] == n):

                parentNode.values = parentNode.values[:i] + \

                    [value] + parentNode.values[i:]

                parentNode.keys = parentNode.keys[:i +

                                                  1] + [ndash] + parentNode.keys[i + 1:]

                if (len(parentNode.keys) > parentNode.order):

                    parentdash = Node(parentNode.order)

                    parentdash.parent = parentNode.parent

                    mid = int(math.ceil(parentNode.order / 2)) - 1

                    parentdash.values = parentNode.values[mid + 1:]

                    parentdash.keys = parentNode.keys[mid + 1:]

                    value\_ = parentNode.values[mid]

                    if (mid == 0):

                        parentNode.values = parentNode.values[:mid + 1]

                    else:

                        parentNode.values = parentNode.values[:mid]

                    parentNode.keys = parentNode.keys[:mid + 1]

                    for j in parentNode.keys:

                        j.parent = parentNode

                    for j in parentdash.keys:

                        j.parent = parentdash

                    self.insert\_in\_parent(parentNode, value\_, parentdash)

    def delete(self, value, key):

        node\_ = self.search(value)

        temp = 0

        for i, item in enumerate(node\_.values):

            if item == value:

                temp = 1

                if key in node\_.keys[i]:

                    if len(node\_.keys[i]) > 1:

                        node\_.keys[i].pop(node\_.keys[i].index(key))

                    elif node\_ == self.root:

                        node\_.values.pop(i)

                        node\_.keys.pop(i)

                    else:

                        node\_.keys[i].pop(node\_.keys[i].index(key))

                        del node\_.keys[i]

                        node\_.values.pop(node\_.values.index(value))

                        self.deleteEntry(node\_, value, key)

                else:

                    print("Value not in Key")

                    return

        if temp == 0:

            print("Value not in Tree")

            return

    def deleteEntry(self, node\_, value, key):

        if not node\_.check\_leaf:

            for i, item in enumerate(node\_.keys):

                if item == key:

                    node\_.keys.pop(i)

                    break

            for i, item in enumerate(node\_.values):

                if item == value:

                    node\_.values.pop(i)

                    break

        if self.root == node\_ and len(node\_.keys) == 1:

            self.root = node\_.keys[0]

            node\_.keys[0].parent = None

            del node\_

            return

        elif (len(node\_.keys) < int(math.ceil(node\_.order / 2)) and node\_.check\_leaf == False) or (len(node\_.values) < int(math.ceil((node\_.order - 1) / 2)) and node\_.check\_leaf == True):

            is\_predecessor = 0

            parentNode = node\_.parent

            PrevNode = -1

            NextNode = -1

            PrevK = -1

            PostK = -1

            for i, item in enumerate(parentNode.keys):

                if item == node\_:

                    if i > 0:

                        PrevNode = parentNode.keys[i - 1]

                        PrevK = parentNode.values[i - 1]

                    if i < len(parentNode.keys) - 1:

                        NextNode = parentNode.keys[i + 1]

                        PostK = parentNode.values[i]

            if PrevNode == -1:

                ndash = NextNode

                value\_ = PostK

            elif NextNode == -1:

                is\_predecessor = 1

                ndash = PrevNode

                value\_ = PrevK

            else:

                if len(node\_.values) + len(NextNode.values) < node\_.order:

                    ndash = NextNode

                    value\_ = PostK

                else:

                    is\_predecessor = 1

                    ndash = PrevNode

                    value\_ = PrevK

            if len(node\_.values) + len(ndash.values) < node\_.order:

                if is\_predecessor == 0:

                    node\_, ndash = ndash, node\_

                ndash.keys += node\_.keys

                if not node\_.check\_leaf:

                    ndash.values.append(value\_)

                else:

                    ndash.nextKey = node\_.nextKey

                ndash.values += node\_.values

                if not ndash.check\_leaf:

                    for j in ndash.keys:

                        j.parent = ndash

                self.deleteEntry(node\_.parent, value\_, node\_)

                del node\_

            else:

                if is\_predecessor == 1:

                    if not node\_.check\_leaf:

                        ndashpm = ndash.keys.pop(-1)

                        ndashkm\_1 = ndash.values.pop(-1)

                        node\_.keys = [ndashpm] + node\_.keys

                        node\_.values = [value\_] + node\_.values

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashkm\_1

                                break

                    else:

                        ndashpm = ndash.keys.pop(-1)

                        ndashkm = ndash.values.pop(-1)

                        node\_.keys = [ndashpm] + node\_.keys

                        node\_.values = [ndashkm] + node\_.values

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashkm

                                break

                else:

                    if not node\_.check\_leaf:

                        ndashp0 = ndash.keys.pop(0)

                        ndashk0 = ndash.values.pop(0)

                        node\_.keys = node\_.keys + [ndashp0]

                        node\_.values = node\_.values + [value\_]

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashk0

                                break

                    else:

                        ndashp0 = ndash.keys.pop(0)

                        ndashk0 = ndash.values.pop(0)

                        node\_.keys = node\_.keys + [ndashp0]

                        node\_.values = node\_.values + [ndashk0]

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndash.values[0]

                                break

                if not ndash.check\_leaf:

                    for j in ndash.keys:

                        j.parent = ndash

                if not node\_.check\_leaf:

                    for j in node\_.keys:

                        j.parent = node\_

                if not parentNode.check\_leaf:

                    for j in parentNode.keys:

                        j.parent = parentNode

def printTree(tree):

    lst = [tree.root]

    level = [0]

    leaf = None

    flag = 0

    lev\_leaf = 0

    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):

        x = lst.pop(0)

        lev = level.pop(0)

        if (x.check\_leaf == False):

            for i, item in enumerate(x.keys):

                print(item.values)

        else:

            for i, item in enumerate(x.keys):

                print(item.values)

            if (flag == 0):

                lev\_leaf = lev

                leaf = x

                flag = 1

record\_len = 3

bplustree = BplusTree(record\_len)

p='y'

while p == 'y':

    print('1.insert\n2.delete\n3.search')

    a = input("Enter your choice: ")

    if a == '1':

        b = int(input("Enter how many inputs: "))

        l1=[]

        for i in range(b):

            c = int(input("enter the number: "))

            l1.append(c)

        for i in l1:

            bplustree.insert(str(i),str(i))

        printTree(bplustree)

    elif a=='2':

        d = int(input("Enter the number you want to delete: "))

        bplustree.delete(str(d),str(d))

        print("now, output is ")

        printTree(bplustree)

    elif a =='3':

        e = int(input("enter the number to be searched: "))

        if (bplustree.find(str(e),str(e))):

            print("Element found")

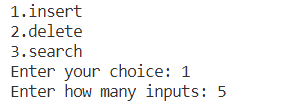
        print("Element not found!!")

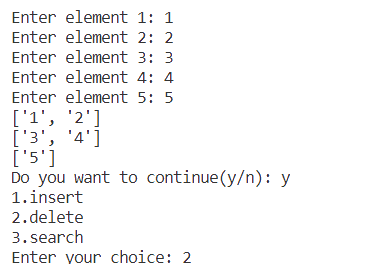
    else:

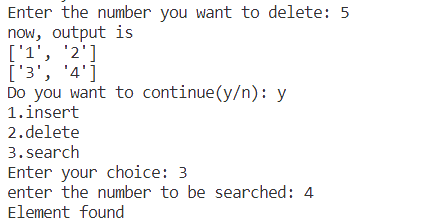
        raise Exception("Invalid input!!")

    p = input("Do you want to continue(y/n): ")

**Output:**

****

****

****

**Time Complexity and Algorithm Analysis:**

**Time Complexity:**

• Search: O(log n)

• Insertion: O(log n)

• Deletion: O(log n)

**Algorithm Analysis:**

• Balanced Structure: B+ trees maintain a balanced structure, ensuring logarithmic

height for efficient operations.

• Large Fanout: B+ trees have a high fanout, reducing the tree height and minimizing

disk I/O.

• Sequential Access: B+ trees leverage sequential access for range queries and

disk operations.

• Disk Block Utilization: B+ trees optimize disk block usage for efficient

storage and retrieval.

**Result:**

Thus, B+ tree has been implemented successfully.

**Implementation of state space search algorithms**

**Aim:**

To create a state space search tree and implement searching algorithms like BFS and DFS.

**Problem Description:**

A state space search tree is a tree that represents all possible feasible solutions to a problem.

**Algorithm:**

**BFS (Breadth-First Search):**

1. Create an empty queue and a visited set.
2. Enqueue the starting vertex or node into the queue.
3. Mark the starting vertex as visited.
4. Enter a loop until the queue is empty.
5. Dequeue a vertex from the front of the queue.
6. Process the dequeued vertex (e.g., print or perform any desired operation).
7. Get all adjacent vertices of the dequeued vertex.
8. For each adjacent vertex, if it is not visited, enqueue it into the queue and mark it as visited.
9. Repeat steps 5-8 until the queue is empty.

**DFS (Depth-First Search):**

1. Create a visited set.
2. Choose a starting vertex or node.
3. Mark the starting vertex as visited.
4. Process the starting vertex (e.g., print or perform any desired operation).
5. Get all adjacent vertices of the starting vertex.
6. For each adjacent vertex, if it is not visited, recursively call the DFS function with the adjacent vertex as the new starting vertex.
7. Repeat steps 4-6 for each unvisited adjacent vertex of the starting vertex.

**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

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

    if node not in visited:

        visited.append(node)   #appending nodes to visited

        for n in graph[node]:

            dfs(graph,n, visited)  #recursive calling of dfs

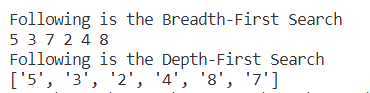
    return visited

print("\nFollowing is the Depth-First Search")

visited = dfs(graph,'5', [])

print(visited)

**Output:**

****

**Time Complexity AND Algorithm Analysis:**

State space search algorithms are used to explore the state space of a problem in order to find a solution. The time complexity and algorithm analysis of state space search algorithms can vary depending on the specific algorithm being used. A few algorithms used are:

• Breadth-First Search (BFS): Time complexity is typically O(b^d), where b is the branching factor and d is the depth of the solution.

• Depth-First Search (DFS): Time complexity is typically O(b^m), where b is the branching factor and m is the maximum depth of the state space.

• A\* Search: Time complexity varies depending on the heuristic used, but in the worst case, it can be exponential

**Result:**

Thus, the state space search algorithms Breadth First Search and Depth First Search has been executed successfully.

**Implementation of divide-and-conquer algorithm**

**for closest-pairs problem**

**Aim:**

To find the closest pair from a given set of points.

**Problem Description:**

We will use a divide and conquer algorithm to solve this problem by dividing it into multiple subproblems and finding an optimized solution.

**Algorithm:**

* Sort the points based on their x-coordinate in ascending order.
* Divide the sorted points into two halves using the vertical line that passes through the median x-coordinate.
* Recursively find the closest pair of points in each half.
* Take the minimum of the two minimum distances obtained from the recursive calls and store it as "minDist."
* Consider the strip of points that lie within a distance of "minDist" from the vertical line.
* Sort these points by their y-coordinate in ascending order.
* Iterate through each point in the strip and compare it with the next 7 points in the sorted order.
* Compute the distance between these pairs of points.
* If any pair has a distance smaller than "minDist," update "minDist" accordingly.
* Return "minDist" as the smallest distance between any two points in the given set.

**Code:**

import math

class Point:  #point class to define a point

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

def compareX(a, b): #comparing x coordinates

    p1 = a

    p2 = b

    return (p1.x - p2.x)

def compareY(a, b):  #comparing y cooordinates

    p1 = a

    p2 = b

    return (p1.y - p2.y)

def dist(p1, p2):   #distance between two points

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

def bruteForce(P, n):   #bruteforce method to find closest distance

    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):  #find the closest strip after

    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):  #divide and conquer method to find closest pair

    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)

while True:

    P = []

    for i in range(int(input("Enter the total no. of points:"))):

        x = int(input("Enter the x of Point %s:"%(i+1)))

        y = int(input("Enter the y of Point %s:"%(i+1)))

        P.append(Point(x,y))

    n = len(P)

    c =  closest(P, n)

    print("The smallest distance is:",c)

    for i in range(n):

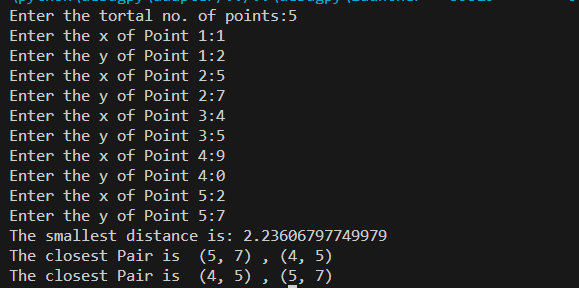
        for j in range(n):

            if dist(P[i],P[j]) == c:

                print("The closest Pair is",(P[i].x,P[i].y),",",(P[j].x,P[j].y))

                break

**Output:**

****

**Time Complexity:**

* Time complexity is O(n log n).
* Sorting the points: O(n log n)
* Recursive calls: T(n/2) each (where T is the time complexity of the
* algorithm)
* Merging and finding the minimum distance: O(n)
* Overall recurrence relation: T(n) = 2T(n/2) + O(n)
* By the Master Theorem, T(n) = O(n log n)

**Algorithm Analysis:**

The Closest Pair algorithm efficiently solves the closest pair problem by dividing the points and recursively finding the closest pairs in smaller subsets. It reduces the number of distance calculations by considering only relevant points within a strip. With a time complexity of O(n log n), the algorithm demonstrates theeffectiveness of divide and conquer in solving geometric problems.

**Result:**

Thus, closest pair algorithm using divide and conquer has been implemented successfully.

**Implementation of Huffman coding**

**Aim:**

To generate Huffman code for a given sequence using a Huffman tree.

**Problem Description:**

A Huffman code is a variable-length data sequence that minimizes the cost of encoding a given sequence with maximum efficiency. A Huffman tree is implemented to generate the same.

**Algorithm:**

* 1. Calculate the frequency of occurrence of each symbol or character in the given data.
  2. Create a leaf node for each symbol or character and assign its frequency as its weight.
  3. Create a min-heap (priority queue) and insert all the leaf nodes into it.
  4. While there is more than one node in the heap:
  5. Extract the two nodes with the minimum frequencies from the heap.
  6. Create a new internal node with a weight equal to the sum of the extracted nodes' frequencies.
  7. Make the first extracted node the left child and the second extracted node the right child of the new node.
  8. Insert the new node back into the heap.
  9. The remaining node in the heap is the root of the Huffman tree.
  10. Traverse the Huffman tree from the root. Assign '0' to the left branch and '1' to the right branch at each step.
  11. Assign unique binary codes to each symbol or character based on the path from the root to that symbol in the Huffman tree.
  12. Encode the original data by replacing each symbol with its corresponding Huffman code.

**Code:**

class Symbol:

    def \_\_init\_\_(self,symbol,freq):

        self.symbol = symbol

        self.freq = freq

        self.code = " "

        self.left = None

        self.right = None

    def \_\_lt\_\_(self,nxt):

        return self.freq < nxt.freq

def printNodes(node, val=''):

    newVal = val + str(node.code)

    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}")

def sorting(data):

    data = sorted(data,key=lambda symbol:symbol.freq)

    return data

data = []

sym = []

while True:

    for i in range(int(input("Enter the Total no.of symbols:"))):

        s = input("Enter the symbol:")

        f = eval(input("Enter the frequency:"))

        symbol = Symbol(s,f)

        data.append(symbol)

        sym.append(symbol)

    data = sorting(data)

    while len(data) > 1:

             first = data.pop(0)

             second = data.pop(0)

             first.code = '0'

             second.code = '1'

             newnode = Symbol(first.symbol+second.symbol,first.freq+second.freq)

             newnode.left = first

             newnode.right = second

             data.append(newnode)

    printNodes(data[0])

**Output:**

**A screenshot of a computer

Description automatically generated**

**TIME COMPLEXITY AND ALGORITH ANALYSIS FOR HUFFMAN CODING**

**USING GREEDY APPROACH:**

**Time Complexity Analysis:** The time complexity of the Huffman coding algorithm depends on the construction of the Huffman tree and the encoding process.

* Construction of Huffman Tree:
* Building the frequency table: O(n), where n is the number of characters in the input.
* Constructing the Huffman tree: O(n log n), where n is the number of characters.
* Encoding Process:
* Generating the Huffman codes: O(n), where n is the number of characters.
* Encoding the input using the Huffman codes: O(m), where m is the length of the input.
* Overall, the time complexity of Huffman coding is O((n + m) log n), where n is
* the number of characters and m is the length of the input.

**Algorithm Analysis:**

* **Optimal Prefix Codes:** Huffman coding guarantees the construction of an

optimal prefix code, where no code is a prefix of another code. This property ensures unique decodability of the encoded data.

* **Variable-Length Codes:** Huffman coding assigns variable-length codes,

with shorter codes for more frequent characters. This results in efficient

compression, as common characters are represented by fewer bits.

* **Lossless Compression:** Huffman coding is a lossless compression

algorithm, meaning that the original data can be perfectly reconstructed

from the encoded data without any loss of information.

* **Greedy Approach**: Huffman coding uses a greedy approach to construct

the optimal prefix code. It builds the Huffman tree by iteratively

combining the two characters with the lowest frequencies until a

complete tree is formed.

* **Compression Ratio:** Huffman coding typically achieves good compression

ratios for text and data with non-uniform character frequencies. The

actual compression ratio depends on the frequency distribution of thecharacters in the input.

**Result:**

Thus, Huffman coding computation has been done successfully.

**Implementation of disjoint sets and Kruskal’s algorithm**

**Aim:**

To implement Kruskal's algorithm using the disjoint-set data structure.

**Problem Description:**

The disjoint-set data structure is used to divide a graph into disjoint sets and makes it easy to detect cycles. Kruskal's algorithm is used to construct a minimum spanning tree.

**Algorithm:**

* + Initialize an empty set to store the edges of the minimum spanning tree (MST).
  + Sort all the edges of the graph in non-decreasing order of their weights.
  + Iterate through the sorted edges in ascending order of their weights.
  + For each edge, check if including it in the MST creates a cycle. You can use a disjoint-set data structure, such as Union-Find, to efficiently check for cycles.
  + If including the edge does not create a cycle, add it to the MST set.
  + Repeat steps 4 and 5 until all the vertices are included in the MST or all the edges have been considered.

**Code:**

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = []

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    def find(self, parent, i):

        if parent[i] != i:

            parent[i] = self.find(parent, parent[i])

        return parent[i]

    def union(self, parent, rank, x, y):

        if rank[x] < rank[y]:

            parent[x] = y

        elif rank[x] > rank[y]:

            parent[y] = x

        else:

            parent[y] = x

            rank[x] += 1

    def KruskalMST(self):

        i,e = 0

        result = []

        self.graph = sorted(self.graph,

                            key=lambda item: item[2])

        parent = []

        rank = []

        for node in range(self.V):

            parent.append(node)

            rank.append(0)

        while e < self.V - 1:

            u, v, w = self.graph[i]

            i = i + 1

            x = self.find(parent, u)

            y = self.find(parent, v)

            if x != y:

                e = e + 1

                result.append([u, v, w])

                self.union(parent, rank, x, y)

        minimumCost = 0

        print("Edges in the constructed MST")

        for u, v, weight in result:

            minimumCost += weight

            print("%d -- %d == %d" % (u, v, weight))

        print("Minimum Spanning Tree", minimumCost)

while True:

    g = Graph(int(input("enter the toatal no. of nodes:")))

    yes = True

    while yes:

        g.addEdge(int(input("enter the source:")), int(input("enter the destination:")), int(input("enter the weight:")))

        op = input("\n1.add more edges,\n2.exit")

        if op == "1":

           yes = True

        if op == "2":

            yes = False

    g.KruskalMST(int(input("Enter the source:")))

**Output:**

**A black screen with white text

Description automatically generated with low confidence**

**Time Complexity:**

* Sorting Edges: Sorting the edges by weight takes O(E log E) time, where E is the number of edges.
* Union-Find Operations: Performing union and find operations on the disjoint-set data structure has an average time complexity of O(log V), where V is the number of vertices.
* Considering the sorting and union-find operations, the overall time complexity of Kruskal's Algorithm is O(E log E) or O(E log V), depending on the graph structure.

**Algorithm Analysis:**

Kruskal's Algorithm is known for its simplicity and efficiency in finding MSTs. It guarantees an MST with the minimum total weight and works well fordense and sparse graphs. The greedy strategy of selecting minimum-weight edges ensures the optimality of the resulting MST. The algorithm is widely used due to its straightforward implementation andability to handle various graph representations. It provides an efficient solution for finding minimum spanning trees and related optimization problems.

**Result:**

Thus, disjoint sets and Kruskal’s algorithms has been implemented successfully.

**Implementation of dynamic programming algorithms**

**Computing Binomial Coefficients**

**Aim:**

To find the binomial coefficient using dynamic programming.

**Problem Description:**

Dynamic programming is used to solve the problem efficiently by dividing it into subproblems and avoiding overlapping subproblems by storing the results in a table.

**Algorithm:**

* + We create a 2D array dp with dimensions (n+1) x (k+1) to store the coefficients. The entry dp[i][j] represents the binomial coefficient

C(i, j).

* + We initialize the base cases: C(n, 0) = C(n, n) = 1. So,

we set dp[i][0] = 1 and dp[i][i] = 1 for all i in the range 0 to n.

* + We calculate the coefficients using the recurrence relation C(n, k) = C(n-1, k-1) + C(n-1, k). We iterate over i from 1 to n and j from 1 to min(i, k) (since choosing more elements than available is not possible).
  + Finally, we return the computed binomial coefficient dp[n][k].

**Code:**

n=int(input())  #Taking inputs for n and k

k=int(input())

c=[[0 for i in range(k+1)] for i in range(n+1)]  #creating an array of size n\*k

def binomial(n,k):

    if k==0 or k==n:  #returning 1 if k=0 or k=n

        c[n][k]=1

        return c[n][k]

    else:

        if c[n][k]:         #returning the value if present in array to avoid overlapping subproblems

            return c[n][k]

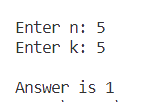
        else:

            c[n][k]=binomial(n-1,k-1)+binomial(n-1,k)

            return c[n][k]

print("answer is",binomial(n,k))

**Output:**

****

**Result:**

Thus, binomial coefficient using dynamic programming has been executed successfully.

**Implementation of dynamic programming algorithms**

**Bellman-Ford algorithm**

**Aim:**

To implement the Bellman-Ford Algorithm using dynamic programming (DP).

**Problem Description:**

The Bellman-Ford Algorithm is a single-source shortest path algorithm used to find the shortest path from a vertex to all other vertices. It can handle cycles and negative weights in the graph.

**Algorithm:**

* + Initialize the distance of the source vertex to 0 and the distance of all other vertices to infinity.
  + Iterate over all vertices in the graph |V| - 1 times, where |V| is the total number of vertices. In each iteration, relax all the edges in the graph.
  + Relaxing an edge means updating the distance of the destination vertex if a shorter path is found. For each edge (u, v) with weight w, if the distance from the source vertex to u plus the weight of the edge is less than the current distance of v, update the distance of v to the new shorter distance.
  + After |V| - 1 iterations, all shortest paths from the source vertex have been found if there are no negative cycles in the graph. If there is a negative cycle, then the algorithm can detect it in the next step.
  + Run an additional iteration over all edges in the graph. If the distance of the destination vertex can still be updated, it means that there is a negative cycle in the graph. In this case, the algorithm cannot find a single shortest path since the negative cycle can be traversed indefinitely to reduce the path length.
  + If no negative cycles are detected, the algorithm has successfully found the shortest path from the source vertex to all other vertices.

**Code:**

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices  # No. of vertices

        self.graph = []

    # function to add an edge to graph

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    #printing the solution

    def printArr(self, dist):

        print("Vertex Distance from Source")

        for i in range(self.V):

            print("{0}\t\t{1}".format(i, dist[i]))

    def BellmanFord(self, src):

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

        dist[src] = 0

        for \_ in range(self.V - 1):

            # queue

            for u, v, w in self.graph:

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

                    dist[v] = dist[u] + w

        for u, v, w in self.graph:

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

                print("Graph contains negative weight cycle")

                return

        self.printArr(dist)

# Driver's code

if \_\_name\_\_ == '\_\_main\_\_':

    g = Graph(5)

    g.addEdge(0, 1, -1)

    g.addEdge(0, 2, 4)

    g.addEdge(1, 2, 3)

    g.addEdge(1, 3, 2)

    g.addEdge(1, 4, 2)

    g.addEdge(3, 2, 5)

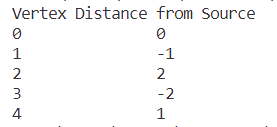
    g.addEdge(3, 1, 1)

    g.addEdge(4, 3, -3)

    # function call

    g.BellmanFord(0)

**Output:**



**TIME COMPLEXITY AND ALGORITHM ANALYSIS:**

**Time Complexity:**

The time complexity of the Bellman-Ford algorithm using

dynamic programming is O(V \* E), where V is the number of vertices and E is the

number of edges in the graph. In the worst case, the algorithm performs V-1

passes over all the edges to find the shortest paths.

**Algorithm Analysis:**

1. Initialization: The algorithm initializes the distance estimates for all

vertices to infinity, except for the source vertex, which is set to 0. This step

takes O(V) time.

2. Relaxation: The algorithm performs V-1 passes over all the edges to relax

them. In each pass, it checks if there is a shorter path to a vertex than the

current distance estimate. If so, it updates the distance estimate. This step

takes O(E) time in each pass.

3. Negative Cycle Detection: After V-1 passes, the algorithm performs one

additional pass to detect negative cycles. If there are still updates to the

distance estimates in this pass, it indicates the presence of a negative

cycle. This step takes O(E) time.

Overall, the Bellman-Ford algorithm using dynamic programming has a time

complexity of O(V \* E).

**Result:**

Thus, bellman ford algorithm using dynamic programming has been computed successfully

**Implementation of backtracking algorithms to solve n-Queens problem**

**Aim:**

To solve the N-queens problem using backtracking.

**Problem Description:**

The N-queens problem is an NP-hard problem where N queens should be placed on a chessboard such that no queens can attack each other.

**Algorithm:**

* + Initialize an empty chessboard of size N x N.
  + Start with the leftmost column and place a queen in the first row of that column.
  + Move to the next column and place a queen in the first row of that column.
  + Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
  + If all N queens have been placed, print the solution.
  + If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
  + Remove the queen from the previous column and move it down one row.
  + Repeat steps 4-7 until all possible configurations have been tried.

**Code:**

def printSolution(board,N):

    for i in range(N):

        for j in range(N):

            if board[i][j] == 1:

                print("Q",end=" ")

            else:

                print(".",end=" ")

        print()

def isSafe(board, row, col,N):

    for i in range(col):

        if board[row][i] == 1:

            return False

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

                    range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    for i, j in zip(range(row, N, 1),

                    range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    return True

def solveNQUtil(board, col,N):

    if col >= N:

        return True

    for i in range(N):

        if isSafe(board, i, col,N):

            board[i][col] = 1

            if solveNQUtil(board, col + 1,N) == True:

                return True

            board[i][col] = 0

    return False

def solveNQ():

    n = int(input("Enter the value of n:"))

    board = [[0 for i in range(n)] for i in range(n)]

    if solveNQUtil(board, 0,n) == False:

        print("Solution does not exist")

        return False

    printSolution(board,n)

    return True

solveNQ()

**Output:**

**A picture containing text, screenshot, font, black

Description automatically generated**

**TIME COMPLEXITY:**

Generating all possible configurations: The number of possible configurations of placing N queens on an NxN chessboard is N!. Therefore, the time complexity of generating all possible configurations is O(N!)

**ALGORITHM ANALYSIS:**

Backtracking and recursion: The backtracking algorithm used to solve the NQueens problem involves recursive calls. In the worst-case scenario, we may have to explore all possible configurations.

**Result:**

Thus, n queens’ problem has been solved using backtracking.

**Implementation of backtracking algorithms**

**Hamiltonian circuit problem**

**Aim:**

To implement the Hamiltonian circuit problem using backtracking.

**Problem description:**

The Hamiltonian circuit problem involves finding a path in a given graph where all vertices are visited exactly once, and the path starts and ends at the same vertex.

**Algorithm:**

* + Start with an empty path and an empty set of visited vertices.
  + Choose a starting vertex arbitrarily and add it to the path and the set of visited vertices.
  + If all vertices are visited and the current vertex has an edge to the starting vertex, then we have found a Hamiltonian circuit. Print the path and terminate the algorithm.
  + If the current vertex has unvisited neighbors, choose an unvisited neighbor and add it to the path and the set of visited vertices.
  + Recursively call the algorithm with the new current vertex.
  + If the recursive call returns false, remove the last vertex from the path and the set of visited vertices.
  + Repeat steps 4-6 for all unvisited neighbors of the current vertex.
  + If no unvisited neighbor leads to a Hamiltonian circuit, return false.

**Code:**

g={"A":["C","B","D"],"B":["A","C","D"],"C":["A","B"],"D":["A","B"]}

a=[]

def hamiltonian\_circuit(g,l):

    for i in g:

        if i==l[-1]:

            for j in g[i]:

                if j not in l:

                    if len(l)+1==len(g):

                        l.append(j)

                        print("Hamiltonian circuit has been found and it is",l)

                        quit()

                    else:

                        m=l.copy()

                        m.append(j)

                        hamiltonian\_circuit(g,m)

hamiltonian\_circuit(g,["A"])

**Output:**



**TIME COMPLEXITY:**

* The Hamiltonian Circuit problem is known to be NP-complete, which means that there is no known polynomial-time algorithm to solve it for all instances. As a result, the time complexity of finding a Hamiltonian Circuit in a graph is exponential.
* The Hamiltonian Circuit problem is computationally challenging, and finding an optimal solution for all instances is believed to be intractable in terms of time complexity.

**ALGORITHM ANALYSIS:**

The brute-force approach to solve the Hamiltonian Circuit problem involves

generating all possible permutations of the vertices and checking each

permutation to see if it forms a Hamiltonian Circuit. The number of

permutations is factorial, which grows very quickly with the number of vertices.

Hence, the time complexity of the brute-force approach is O(n!), where n is the

number of vertices in the graph.

There are also more efficient algorithms and heuristics for solving the

Hamiltonian Circuit problem, such as backtracking algorithms, dynamic

programming, and branch-and-bound techniques. These algorithms can

improve the average-case performance for certain types of graphs or provide

approximate solutions. However, their worst-case time complexity remains

exponential.

**Result:**

Thus, Hamiltonian circuit has been solved using backtracking successfully.

**Implementation of iterative improvement strategy**

**for stable marriage problem**

**Aim:**

To implement the stable marriage problem using an iterative improvement strategy.

**Problem Description:**

The stable marriage problem is an iterative improvement problem where men and women are engaged based on their preference order. If a man and woman are already engaged, the woman can be engaged based on her preference, and the process repeats.

**Algorithm:**

* Initially, all men and women are free, and the matching is empty. The following steps are performed iteratively:
  + Randomly select a man and let him propose to the first woman on his preference list who hasn't rejected him so far.
  + If the woman is free, she accepts his proposal, and they become a pair in the matching. If she is not free, she compares her current match with the proposing man. If she prefers the proposing man to her current match, she accepts the proposal and replaces her current match. This way, a new pair is formed. If she doesn't prefer the proposing man, she rejects the proposal, and the proposing man remains unmatched.
* These two steps are repeated until no man remains unmatched. Since no two men can be matched with the same woman, all the women have their matches when the algorithm terminates.

**Code:**

N = 4

def womenPrefersM1OverM(prefer, w, m, m1):

    for i in range(N):

        if (prefer[w][i] == m1):

            return True

        if (prefer[w][i] == m):

            return False

def stableMarriage(prefer):

    wPartner = [-1 for i in range(N)]

    mFree = [False for i in range(N)]

    freeCount = N

    # While there are free men

    while (freeCount > 0):

        # Pick the first free man (we could pick any)

        m = 0

        while (m < N):

            if (mFree[m] == False):

                break

            m += 1

        i = 0

        while i < N and mFree[m] == False:

            w = prefer[m][i]

            if (wPartner[w - N] == -1):

                wPartner[w - N] = m

                mFree[m] = True

                freeCount -= 1

            else:

                m1 = wPartner[w - N]

                if (womenPrefersM1OverM(prefer, w, m, m1) == False):

                    wPartner[w - N] = m

                    mFree[m] = True

                    mFree[m1] = False

            i += 1

    print("Woman\tMan")

    for i in range(N):

        print(i + N, "\t", wPartner[i])

# Driver Code

choice = [[7, 5, 6, 4],

          [5, 4, 6, 7],

          [4, 5, 6, 7],

          [4, 5, 6, 7],

          [0, 1, 2, 3],

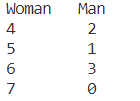
          [0, 1, 2, 3],

          [0, 1, 2, 3],

          [0, 1, 2, 3]]

stableMarriage(choice)

**Output:**

****

**TIME COMPLEXITY:**

The Stable Marriage Problem is typically solved using the Gale-Shapley

algorithm, which has a time complexity of O(n^2), where n is the number of men

or women in the problem.

**TIME COMPLEXITY ANALYSIS:**

* The initialization step takes O(n) time to create and initialize the arrays.
* The main loop executes at most n iterations since each man can make a

proposal to every woman.

* Proposing to a woman and updating the engagement takes O(1) time.
* Therefore, the overall time complexity of the Gale-Shapley algorithm is

O(n^2).

A**LGORITHM ANLAYSIS:**

* **Input:** The algorithm takes as input the preferences of men and women.
* **Initialization:** It initializes the engaged array and men proposals array ,both of size n, to track the current engagements and proposals made by each man, respectively.
* **Main Loop:** The algorithm continues until all men are engaged. Inside the loop, each man proposes to the next preferred woman who has not rejected him yet.
* **Proposal:** If the woman is not engaged, she accepts the proposal, and the engagement is updated. Otherwise, if the woman is already engaged, she compares the current man with her current partner based on her
* **preferences.** If the current man is preferred, the engagement is update by replacing the current partner.
* **Output:** Finally, the algorithm prints the engagements.

**Result:**

Thus, the stable marriage problem has been solved successfully

**Implementation of iterative improvement strategy**

**for max flow problem**

**Aim:**

To implement the max flow algorithm using the iterative improvement strategy.

**Problem Description:**

The max flow algorithm, also known as the Ford-Fulkerson algorithm, is an iterative-based algorithm used to calculate the maximum flow that can pass from the source to the sink while considering capacity constraints.

**Algorithm:**

* + Start with the initial flow set to 0.
  + While there exists an augmenting path from the source to the sink:
    - Find an augmenting path using any path-finding algorithm, such as breadth-first search or depth-first search.
    - Determine the amount of flow that can be sent along the augmenting path, which is the minimum residual capacity along the edges of the path.
    - Increase the flow along the augmenting path by the determined amount.
  + Return the maximum flow.

**Code:**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self, graph):

        self.graph = graph

        self. ROW = len(graph)

    def searching\_algo\_BFS(self, s, t, parent):

        visited = [False] \* (self.ROW)

        queue = []

        queue.append(s)

        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):

                if visited[ind] == False and val > 0:

                    queue.append(ind)

                    visited[ind] = True

                    parent[ind] = u

        return True if visited[t] else False

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

        parent = [-1] \* (self.ROW)

        max\_flow = 0

        while self.searching\_algo\_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

                v = parent[v]

        return max\_flow

graph = [[0, 8, 0, 0, 3, 0],

         [0, 0, 9, 0, 0, 0],

         [0, 0, 0, 0, 7, 2],

         [0, 0, 0, 0, 0, 5],

         [0, 0, 7, 4, 0, 0],

         [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0

sink = 5

print("\nMax Flow: %d " % g.ford\_fulkerson(source, sink))

**Output:**



**TIME COMPLEXITY:**

In the worst case, where capacities are integers, the time complexity can be O(E

\* |f\*|), where E is the number of edges and |f\*| is the maximum flow value. By

using efficient augmenting path search algorithms like Edmonds-Karp (with BFS),

the time complexity can be reduced to O(V \* E^2), where V is the number of

vertices and E is the number of edges.

**ALGORITHM ANALYSIS:**

The Ford-Fulkerson algorithm is commonly used to solve the maximum flow

problem. The time complexity of the algorithm depends on the specific

implementation and the choice of augmenting path search algorithm.

The Ford-Fulkerson algorithm is practical and efficient for many cases, but for

large networks or specialized requirements, other algorithms like the PushRelabel algorithm may offer better performance. The choice of algorithm

depends on the problem instance and efficiency requirements.

**Result:**

Thus, max flow problem has been executed successfully using iterative improvement.

**Implementation of Branch and Bound technique**

**to solve knapsack problem**

**Aim:**

To solve knapsack problem using branch and bound technique

**Problem description:**

A knapsack problem states that given a sack of capacity W ,we have to find the maximum profit that is possible by putting in items whose weights do not exceed the maximum capacity

**Algorithm:**

* Start with an empty knapsack and set the initial upper bound (UB) to zero.
* Create a priority queue (usually implemented as a min-heap) to store partial solutions.
* Compute the bound (B) for the initial empty solution and insert it into the priority queue.
* While the priority queue is not empty:
* Extract the solution with the lowest bound from the priority queue.
* If the bound is less than the current upper bound (UB), skip this solution.
* Otherwise, explore two branches:
* Include the next item and update the weight and value accordingly.
* Exclude the next item and keep the weight and value unchanged.
* Compute the bounds for both branches.
* If the weight of the current solution exceeds the knapsack capacity, skip this solution.
* If the value of the current solution is higher than the current upper bound (UB), update UB.
* Insert the branches into the priority queue.
* When the priority queue is empty, the best solution found is the one with the highest value

**Code:**

a=[]

n=int(input("enter no of items: "))

for i in range(n):

    a.append(eval(input("enter weight value pair of item: ")))

for i in a:

    i.append(i[1]/i[0])

b=sorted(a,key=lambda x:x[2])

b.reverse()

print(b)

max1=int(input("enter maximum capacity: "))

co=0

def knapsack(queue):

    global co

    if co==len(b)-1:

        if queue[1]+b[co][0]<=max1:

            print(queue[0]+b[co][1])

        else:

            print(queue[0])

    else:

        k1=0

        left\_child=[queue[0]+b[co][1],queue[1]+b[co][0]]

        m=left\_child[0]

        n=left\_child[1]

        if left\_child[1]>max1:

            k1=1

        if k1==1:

             right\_child=[queue[0],queue[1]]

             right\_child.append(queue[0]+(max1-queue[1])\*b[co+1][-1])

             co+=1

             knapsack(right\_child)

        else:

            left\_child.append(m+(max1-n)\*b[co+1][-1])

            right\_child=[queue[0],queue[1]]

            right\_child.append(queue[0]+(max1-queue[1])\*b[co+1][-1])

            if left\_child[-1]>right\_child[-1]:

                co+=1

                knapsack(left\_child)

            elif left\_child[-1]==right\_child[-1]:

                co+=1

                knapsack(left\_child)

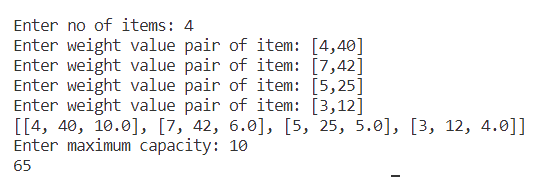
            else:

                co+=1

                knapsack(right\_child)

knapsack([0,0,75])

**Output:**

****

**Time Complexity:**

Exponential time complexity, typically represented as O(2^n), where n is the number of items. The algorithm explores all possible subsets of items to find the optimal solution.

**Algorithm Analysis:**

The Branch and Bound technique efficiently

explores the solution space by branching on each item's inclusion or

exclusion and using a bound function to prune infeasible or

suboptimal solutions. This significantly reduces the search space and

improves efficiency.

**Result:**

Thus, knapsack problem has been solved successfully using branch and bound

**Implementation of Branch and Bound technique**

**to solve Travelling salesman problem**

**Aim:**

To implement the Traveling Salesman Problem (TSP) using branch and bound.

**Problem Description:**

The Traveling Salesman Problem involves finding the minimum cost incurred while visiting all cities exactly once, starting and ending at the same city.

**Algorithm:**

* Start with an initial city as the current city and an empty route.Generate all possible extensions of the current route by adding one city at a time to it. Each extension represents a potential partial tour.
* Assign a lower bound to each partial tour. This lower bound is obtained by calculating the length of the partial tour so far and estimating the length of the remaining tour using a heuristic or lower bound technique (e.g., the length of a minimum spanning tree).
* Prune branches that have a higher cost than the best solution found so far. If the lower bound of a partial tour exceeds the cost of the best solution, discard it.
* Explore the remaining branches recursively by selecting the one with the lowest lower bound and repeating steps 2 to 4.
* When all branches have been explored, the algorithm terminates, and the best solution found during the process is the optimal solution to the TSP.

**Code:**

import math

maxsize = float('inf')

def copyToFinal(curr\_path):

    final\_path[:N + 1] = curr\_path[:]

    final\_path[N] = curr\_path[0]

def firstMin(adj, i):

    min = maxsize

    for k in range(N):

        if adj[i][k] < min and i != k:

            min = adj[i][k]

    return min

def secondMin(adj, i):

    first, second = maxsize, maxsize

    for j in range(N):

        if i == j:

            continue

        if adj[i][j] <= first:

            second = first

            first = adj[i][j]

        elif(adj[i][j] <= second and

            adj[i][j] != first):

            second = adj[i][j]

    return second

def TSPRec(adj, curr\_bound, curr\_weight,

            level, curr\_path, visited):

    global final\_res

    if level == N:

        if adj[curr\_path[level - 1]][curr\_path[0]] != 0:

            curr\_res = curr\_weight + adj[curr\_path[level - 1]]\

                                        [curr\_path[0]]

            if curr\_res < final\_res:

                copyToFinal(curr\_path)

                final\_res = curr\_res

        return

    for i in range(N):

        if (adj[curr\_path[level-1]][i] != 0 and

                            visited[i] == False):

            temp = curr\_bound

            curr\_weight += adj[curr\_path[level - 1]][i]

            if level == 1:

                curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) +

                                firstMin(adj, i)) / 2)

            else:

                curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) +

                                firstMin(adj, i)) / 2)

            if curr\_bound + curr\_weight < final\_res:

                curr\_path[level] = i

                visited[i] = True

                TSPRec(adj, curr\_bound, curr\_weight,

                    level + 1, curr\_path, visited)

            curr\_weight -= adj[curr\_path[level - 1]][i]

            curr\_bound = temp

            visited = [False] \* len(visited)

            for j in range(level):

                if curr\_path[j] != -1:

                    visited[curr\_path[j]] = True

def TSP(adj):

    curr\_bound = 0

    curr\_path = [-1] \* (N + 1)

    visited = [False] \* N

    for i in range(N):

        curr\_bound += (firstMin(adj, i) +

                    secondMin(adj, i))

    curr\_bound = math.ceil(curr\_bound / 2)

    visited[0] = True

    curr\_path[0] = 0

    TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited)

N = 5

adj =[[0,3,1,5,8],

      [3,0,6,7,9],

      [1,6,0,4,2],

      [5,7,4,0,3],

      [8,9,2,3,0]]

# final\_path[] stores the final solution

# i.e. the // path of the salesman.

final\_path = [None] \* (N + 1)

# visited[] keeps track of the already

# visited nodes in a particular path

visited = [False] \* N

# Stores the final minimum weight

# of shortest tour.

final\_res = maxsize

TSP(adj)

print("Minimum cost :", final\_res)

print("Path Taken : ", end = ' ')

for i in range(N + 1):

    print(final\_path[i], end = ' ')

**Output:**

****

**Time Complexity:**

* Factorial time complexity, typically represented as O(n!), where n is the number of cities. The algorithm explores all possible permutations of cities to find the optimal tour.

**Algorithm Analysis:**

* The Branch and Bound technique systematically explores the solution space by branching on unvisited cities and using a lower bound function to estimate minimum tour costs. It efficiently prunes unpromising subproblems, making the search more focused and efficient.

**Result:**

Thus travelling salesman problem has been solved using branch and bound successfully.

**Implementation of approximation algorithms**

**for travelling salesman problem**

**Aim:**

To implement the traveling salesman problem using an approximation algorithm.

**Problem description:**

The traveling salesman problem states that we need to find the least cost incurred while visiting all the cities exactly once, starting and ending at the same city.

**Algorithm:**

* Create a minimum spanning tree (MST) of the given graph representing cities.
* Find all the vertices with odd degrees in the MST.
* Create a minimum-weight perfect matching among these odd-degree vertices.
* Combine the MST and the minimum-weight perfect matching to form a connected graph.
* Find an Eulerian circuit in the resulting graph.
* Convert the Eulerian circuit into a Hamiltonian circuit (TSP tour) by skipping repeated vertices and returning to the starting city.

**Code:**

import math

def distance(city1, city2, city\_distances):

    return city\_distances[city1][city2]

def nearest\_neighbor(city\_distances, starting\_city):

    num\_cities = len(city\_distances)

    tour = [starting\_city]

    remaining\_cities = list(range(num\_cities))

    remaining\_cities.remove(starting\_city)

    while remaining\_cities:

        current\_city = tour[-1]

        nearest\_city = min(remaining\_cities, key=lambda c: distance(current\_city, c, city\_distances))

        tour.append(nearest\_city)

        remaining\_cities.remove(nearest\_city)

    return tour

# Get input from the user

num\_cities = int(input("Enter the number of cities: "))

city\_distances = []

for i in range(num\_cities):

    distances = input(f"Enter the distances from city {i+1} to all other cities separated by spaces: ").split()

    distances = [int(d) for d in distances]

    city\_distances.append(distances)

starting\_city = int(input("Enter the starting city (1 to N): ")) - 1

# Solve TSP using nearest neighbor heuristic

tour = nearest\_neighbor(city\_distances, starting\_city)

total\_distance = sum(distance(tour[i], tour[i+1], city\_distances) for i in range(len(tour)-1))

total\_distance += distance(tour[-1], tour[0], city\_distances)

# Adjust tour and starting\_city index for printing

tour = [city + 1 for city in tour]

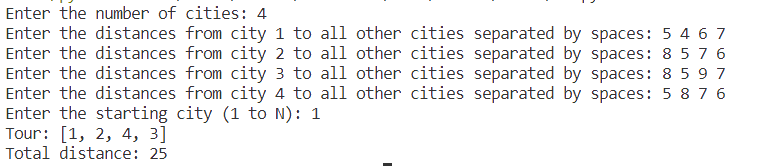
starting\_city += 1

# Print the result

print("Tour:", tour)

print("Total distance:", total\_distance)

**Output:**

****

• **Time Complexity:** Approximation algorithms for the TSP also have polynomial time complexities, typically ranging from O(n^2) to O(n^3) where n is the number of cities.

**• Algorithm Analysis:** These algorithms aim to find tours that are reasonably close to the optimal tours in terms of length or cost. They achieve this by using heuristics and greedy techniques to construct tours iteratively, considering factors like proximity. Although the solutions may not be optimal, they are usually efficient and provide reasonably good approximations.

**Result:**

Thus, travelling salesman problem has been solved succeffully using approximation algorithm

**Implementation of approximation algorithms**

**for knapsack problem**

**Aim:**

To implement the Knapsack problem using approximation.

**Problem Description:**

The Knapsack problem is an NP-hard problem that requires finding the maximum profit possible by sequentially placing items without exceeding the maximum capacity.

**Algorithm:**

* + Calculate the value-to-weight ratio for each item in the knapsack: ratio = value / weight.
  + Sort the items in descending order based on their value-to-weight ratio.
  + Initialize the total value and total weight variables to 0.
  + Iterate through the sorted items:
  + If the weight of the current item is less than or equal to the remaining capacity of the knapsack, include the whole item.
  + Add the value of the current item to the total value.
  + Subtract the weight of the current item from the remaining capacity of the knapsack.
  + If the weight of the current item is greater than the remaining capacity, include a fraction of the item.
  + Calculate the fraction by dividing the remaining capacity by the weight of the current item.
  + Break out of the loop since the knapsack is full.
  + Return the total value as the approximate solution to the Knapsack problem.

**Code:**

def knapsack\_approximation(values, weights, capacity):

    items = list(zip(values, weights))

    items.sort(key=lambda x: x[0] / x[1], reverse=True)

    total\_value = 0

    knapsack = []

    for value, weight in items:

        if weight <= capacity:

            knapsack.append((value, weight))

            total\_value += value

            capacity -= weight

    return knapsack, total\_value

# Get user input for values and weights

values = input("Enter the values (comma-separated): ").split(",")

weights = input("Enter the weights (comma-separated): ").split(",")

# Convert input values and weights to integers

values = [int(value) for value in values]

weights = [int(weight) for weight in weights]

capacity = int(input("Enter the capacity of the knapsack: "))

knapsack, total\_value = knapsack\_approximation(values, weights, capacity)

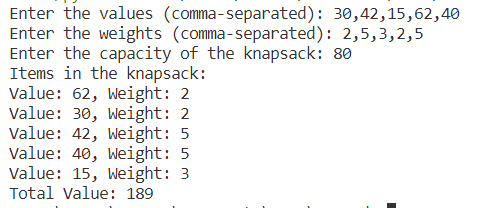
print("Items in the knapsack:")

for value, weight in knapsack:

    print(f"Value: {value}, Weight: {weight}")

print(f"Total Value: {total\_value}")

**Output:**

****

* **Time Complexity:** Approximation algorithms for the Knapsack Problem have polynomial time complexities, typically around O(n^2) where n is the

number of items.

* **Algorithm Analysis:** These algorithms provide efficient solutions by using

heuristics and greedy strategies to select items based on value and weight

ratios. While the solutions may not be optimal, they are generally close

enough to be considered satisfactory.

**Result:**

Thus, knapsack has been solved using approximation algorithm.

**Parallel and Randomized Algorithm**

**Aim:**

To write the Python code to implement Parallel and Randomized algorithms.

**Problem Description:**

The problem is to efficiently calculate the total sum of squares for a large list of numbers using parallel and randomized computation. The algorithm generates a list of numbers, shuffles them randomly, and then divides the list into chunks. Each chunk is processed independently by a separate worker process, calculating the sum of squares. The results from each process are combined to obtain the final total sum of squares. The use of parallel processing and randomization improves the efficiency and introduces variability in the computation.

**Algorithm:**

* Generate a list of numbers.
* Shuffle the list randomly.
* Split the list into chunks based on the number of CPU cores available.
* Assign each chunk to a separate worker process.
* Each worker process calculates the sum of squares for its assigned chunk.
* Combine the results from each worker process by summing them up.
* Output the total sum of squares.

**Code:**

import multiprocessing

import random

def calculate\_sum\_of\_squares(numbers):

    total = 0

    for num in numbers:

        total += num \* num

    return total

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

    num\_processes = multiprocessing.cpu\_count()

    num\_elements = int(input("\nEnter no of elements: "))

    numbers = [random.randint(1, 100) for \_ in range(num\_elements)]

    random.shuffle(numbers)

    chunk\_size = num\_elements // num\_processes

    chunks = [numbers[i:i+chunk\_size] for i in range(0, num\_elements, chunk\_size)]

    pool = multiprocessing.Pool(processes=num\_processes)

    results = pool.map(calculate\_sum\_of\_squares, chunks)

    total\_sum = sum(results)

    print("Total sum of squares:", total\_sum)

**Output:**



**Result:**

Thus, the sum of squares of numbers has been successfully calculated using parallel and randomized algorithm.