**1.**

**Write a program that implements Bubble sort:**

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

# Last i elements are already in place, so we don't need to check them

for j in range(0, n - i - 1):

# Swap if the element found is greater than the next element

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Example usage:

arr = [64, 34, 25, 12, 22, 11, 90]

print("Original array:", arr)

bubble\_sort(arr)

print("Sorted array:", arr)

**2.**

**Write a program that implements insertion sort:**

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i-1

while j >=0 and key < arr[j] :

arr[j+1] = arr[j]

j -= 1

arr[j+1] = key

# Example usage:

arr = [12, 11, 13, 5, 6]

insertion\_sort(arr)

print("Sorted array is:", arr)

**3.**

**Write a program that implements selection sort:**

def selection\_sort(arr):

n = len(arr)

for i in range(n):

# Find the minimum element in the remaining unsorted array

min\_index = i

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

if arr[j] < arr[min\_index]:

min\_index = j

# Swap the found minimum element with the first element

arr[i], arr[min\_index] = arr[min\_index], arr[i]

arr = [64, 25, 12, 22, 11]

selection\_sort(arr)

print("Sorted array is:", arr)

**4.**

**Write a program to implement merge sort**

def merge(arr, p, q, r):

n1 = q - p + 1

n2 = r - q

left = arr[p:q + 1]

right = arr[q + 1:r + 1]

i = j = 0

k = p

while i < n1 and j < n2:

if left[i] <= right[j]:

arr[k] = left[i]

i += 1

else:

arr[k] = right[j]

j += 1

k += 1

while i < n1:

arr[k] = left[i]

i += 1

k += 1

while j < n2:

arr[k] = right[j]

j += 1

k += 1

def merge\_sort(arr, p, r):

if p < r:

q = (p + r) // 2

merge\_sort(arr, p, q)

merge\_sort(arr, q + 1, r)

merge(arr, p, q, r)

arr = [33, 10, 5, 28]

merge\_sort(arr, 0, len(arr) - 1)

print("Sorted Array:", arr)

**5.**

**Write a program to Sort a given set of elements using the Quick sort**

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

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

return i + 1

def quick\_sort(arr, low, high):

if low < high:

pi = partition(arr, low, high)

quick\_sort(arr, low, pi - 1)

quick\_sort(arr, pi + 1, high)

# Example usage:

arr = [10, 7, 8, 9, 1, 5]

quick\_sort(arr, 0, len(arr) - 1)

print("Sorted array is:", arr)

**6.**

**Write a program that implements Linear search.**

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i # Return the index of the target element if found

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

# Example usage:

arr = [4, 2, 7, 1, 9, 5]

target = 7

result = linear\_search(arr, target)

if result != -1:

print(f"Element {target} found at index {result}.")

else:

print(f"Element {target} not found in the array.")

**7.**

**Write a program that implements binary search.**

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid # Return the index of the target element if found

elif arr[mid] < target:

low = mid + 1 # Continue searching in the right half

else:

high = mid - 1 # Continue searching in the left half

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

# Example usage:

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]

target = 7

result = binary\_search(arr, target)

if result != -1:

print(f"Element {target} found at index {result}.")

else:

print(f"Element {target} not found in the array.")

**8.**

**Write a program to implement Binary search tree**

class TreeNode:

def \_\_init\_\_(self, key):

self.val = key

self.left= None

self.right= None

class BST:

def \_\_init\_\_(self):

self.root = None

def insert(self,root,key):

if root is None:

return TreeNode(key)

if key<root.val:

root.left = self.insert(root.left,key)

else:

root.right = self.insert(root.right,key)

return root

def insert\_key(self,key):

self.root= self.insert(self.root,key)

def search(self,root,key):

if root is None or root.val==key:

return root

if key < root.val:

return self.search(root.left,key)

return self.search(root.right, key)

def search\_key(self,key):

return self.search(self.root,key)

def inorder\_traversal(self,root):

if root:

self.inorder\_traversal(root.left)

print(root.val, end = " ")

self.inorder\_traversal(root.right)

def inorder(self):

self.inorder\_traversal(self.root)

bst = BST()

keys = [10, 40, 10, 50, 60, 70]

for key in keys:

bst.insert\_key(key)

print("Inorder traversal of BST:")

bst.inorder()

**9.**

**Write a program to find optimal ordering of matrix multiplication**

import sys

def matrix\_chain\_order(p):

n = len(p) - 1 # Number of matrices

m = [[0] \* n for \_ in range(n)]

s = [[0] \* n for \_ in range(n)]

for l in range(2, n + 1): # Length of chain

for i in range(n - l + 1):

j = i + l - 1

m[i][j] = sys.maxsize

for k in range(i, j):

q = m[i][k] + m[k + 1][j] + p[i] \* p[k + 1] \* p[j + 1]

if q < m[i][j]:

m[i][j] = q

s[i][j] = k

return m, s

def print\_optimal\_parens(s, i, j):

if i == j:

print(f"A{i}", end="")

else:

print("(", end="")

print\_optimal\_parens(s, i, s[i][j])

print\_optimal\_parens(s, s[i][j] + 1, j)

print(")", end="")

# Example usage:

p = [30, 35, 15, 5, 10, 20, 25] # Matrix dimensions: [30x35, 35x15, 15x5, 5x10, 10x20, 20x25]

m, s = matrix\_chain\_order(p)

print("Minimum number of scalar multiplications:", m[0][len(p) - 2])

print("Optimal ordering of matrix multiplication:", end=" ")

print\_optimal\_parens(s, 0, len(p) - 2)

**10. Implement 0/1 Knapsack problem using Dynamic Programming**

def knapsack(weights, values, capacity):

n = len(weights)

# Initialize a table to store the maximum value for each subproblem

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

# Build the table bottom-up

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

for w in range(1, capacity + 1):

# If the current item's weight is greater than the capacity,

# we cannot include it in the knapsack

if weights[i - 1] > w:

dp[i][w] = dp[i - 1][w]

else:

# Otherwise, consider including or excluding the current item

dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])

# Reconstruct the solution

selected\_items = []

w = capacity

for i in range(n, 0, -1):

if dp[i][w] != dp[i - 1][w]:

selected\_items.append(i - 1)

w -= weights[i - 1]

return dp[n][capacity], selected\_items

# Example usage:

weights = [10, 20, 30]

values = [60, 100, 120]

capacity = 50

max\_value, selected\_items = knapsack(weights, values, capacity)

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

print("Selected items:", selected\_items)

**11. Write a program that implements knapsack using greedy**

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

n = len(weights)

# Calculate the value-to-weight ratio for each item

ratios = [(values[i] / weights[i], i) for i in range(n)]

# Sort items by value-to-weight ratio in descending order

ratios.sort(reverse=True)

total\_value = 0

selected\_items = []

for ratio, i in ratios:

print(i)

if weights[i] <= capacity:

# Include the entire item if it fits in the knapsack

total\_value += values[i]

capacity -= weights[i]

selected\_items.append(i)

return total\_value, selected\_items

# Example usage:

weights = [10, 20, 30]

values = [60, 100, 120]

capacity = 50

max\_value, selected\_items = knapsack\_greedy(weights, values, capacity)

print("Maximum value (greedy approach):", max\_value)

print("Selected items:", selected\_items)

1. **Write a program to implement file compression (and un-compression) using Huffman’s algorithm.**

class HuffmanNode:

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

self.char = char

self.freq = freq

self.left = None

self.right = None

def build\_huffman\_tree(text):

frequency = {}

for char in text:

frequency[char] = frequency.get(char, 0) + 1

nodes = [HuffmanNode(char, freq) for char, freq in frequency.items()]

while len(nodes) > 1:

nodes.sort(key=lambda x: x.freq)

left = nodes.pop(0)

right = nodes.pop(0)

merged = HuffmanNode(None, left.freq + right.freq)

merged.left = left

merged.right = right

nodes.append(merged)

return nodes[0]

def build\_huffman\_codes(root, code="", huffman\_codes={}):

if root is None:

return

if root.char is not None:

huffman\_codes[root.char] = code

build\_huffman\_codes(root.left, code + "0", huffman\_codes)

build\_huffman\_codes(root.right, code + "1", huffman\_codes)

def compress(input\_file, output\_file):

with open(input\_file, 'r') as f:

text = f.read()

root = build\_huffman\_tree(text)

huffman\_codes = {}

build\_huffman\_codes(root, "", huffman\_codes)

encoded\_text = ''.join(huffman\_codes[char] for char in text)

with open(output\_file, 'wb') as f:

f.write(int(encoded\_text, 2).to\_bytes((len(encoded\_text) + 7) // 8, byteorder='big'))

def decompress(input\_file, output\_file):

with open(input\_file, 'rb') as f:

encoded\_text = ''.join(format(byte, '08b') for byte in f.read())

root = build\_huffman\_tree(encoded\_text)

huffman\_codes = {}

build\_huffman\_codes(root, "", huffman\_codes)

decoded\_text = ""

node = root

for bit in encoded\_text:

if bit == '0':

node = node.left

else:

node = node.right

if node.char is not None:

decoded\_text += node.char

node = root

with open(output\_file, 'w') as f:

f.write(decoded\_text)

# Example usage:

input\_file = 'input.txt'

compressed\_file = 'compressed.bin'

decompressed\_file = 'decompressed.txt'

compress(input\_file, compressed\_file)

decompress(compressed\_file, decompressed\_file)

**13. Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.**

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

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

self.graph.append([u, v, w])

def find(self, parent, i):

if parent[i] == i:

return i

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

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

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

def kruskal\_mst(self):

result = []

i, e = 0, 0

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)

return result

# Example usage:

cost = [[0, 10, 20], [10, 0, 30], [20, 30, 0]]

g = Graph(len(cost))

for i in range(len(cost)):

for j in range(len(cost[0])):

if cost[i][j] != 0:

g.add\_edge(i, j, cost[i][j])

print("Edges of MST using Kruskal's algorithm:")

print(g.kruskal\_mst())

**14. Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.**

import sys

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for \_ in range(vertices)] for \_ in range(vertices)]

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

self.graph[u][v] = w

self.graph[v][u] = w

def min\_key(self, key, mst\_set):

min\_val = sys.maxsize

min\_index = -1

for v in range(self.V):

if key[v] < min\_val and not mst\_set[v]:

min\_val = key[v]

min\_index = v

return min\_index

def prim\_mst(self):

parent = [-1] \* self.V

key = [sys.maxsize] \* self.V

mst\_set = [False] \* self.V

key[0] = 0

parent[0] = -1

for \_ in range(self.V - 1):

u = self.min\_key(key, mst\_set)

mst\_set[u] = True

for v in range(self.V):

if self.graph[u][v] > 0 and not mst\_set[v] and key[v] > self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

result = []

for i in range(1, self.V):

result.append((parent[i], i, self.graph[i][parent[i]]))

return result

# Example usage:

cost = [[0, 10, 20], [10, 0, 30], [20, 30, 0]]

g = Graph(len(cost))

for i in range(len(cost)):

for j in range(len(cost[0])):

if cost[i][j] != 0:

g.add\_edge(i, j, cost[i][j])

print("Edges of MST using Prim's algorithm:")

print(g.prim\_mst())

**15. Write a program to implements Dijkstra’s algorithm.**

import sys

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for \_ in range(vertices)] for \_ in range(vertices)]

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

self.graph[u][v] = w

def min\_distance(self, dist, spt\_set):

min\_val = sys.maxsize

min\_index = -1

for v in range(self.V):

if dist[v] < min\_val and not spt\_set[v]:

min\_val = dist[v]

min\_index = v

return min\_index

def dijkstra(self, src):

dist = [sys.maxsize] \* self.V

dist[src] = 0

spt\_set = [False] \* self.V

for \_ in range(self.V):

u = self.min\_distance(dist, spt\_set)

spt\_set[u] = True

for v in range(self.V):

if self.graph[u][v] > 0 and not spt\_set[v] and dist[v] > dist[u] + self.graph[u][v]:

dist[v] = dist[u] + self.graph[u][v]

return dist

# Example usage:

g = Graph(9)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 7, 8)

g.add\_edge(1, 2, 8)

g.add\_edge(1, 7, 11)

g.add\_edge(2, 3, 7)

g.add\_edge(2, 8, 2)

g.add\_edge(2, 5, 4)

g.add\_edge(3, 4, 9)

g.add\_edge(3, 5, 14)

g.add\_edge(4, 5, 10)

g.add\_edge(5, 6, 2)

g.add\_edge(6, 7, 1)

g.add\_edge(6, 8, 6)

g.add\_edge(7, 8, 7)

src = 0

print(f"Shortest distances from vertex {src} to all other vertices:")

print(g.dijkstra(src))

**16. Write a program to implement All-Pairs Shortest Paths Problem using Floyd's**

**algorithm.**

INF = float('inf')

def floyd\_warshall(graph):

V = len(graph)

dist = [[0]\*V for \_ in range(V)]

for i in range(V):

for j in range(V):

dist[i][j] = graph[i][j]

for k in range(V):

for i in range(V):

for j in range(V):

if dist[i][k] + dist[k][j] < dist[i][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

# Example usage:

graph = [

[0, 5, INF, 10],

[INF, 0, 3, INF],

[INF, INF, 0, 1],

[INF, INF, INF, 0]

]

shortest\_paths = floyd\_warshall(graph)

print("Shortest paths between all pairs of vertices:")

for row in shortest\_paths:

print(row)

1. **Find a subset of a given set S = {s1, s2 ,..,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.**

def find\_subset\_util(arr, subset, target, idx, result):

if target == 0:

result.append(subset[:])

return

if idx >= len(arr) or target < 0:

return

# Include current element

subset.append(arr[idx])

find\_subset\_util(arr, subset, target - arr[idx], idx + 1, result)

# Exclude current element

subset.pop()

find\_subset\_util(arr, subset, target, idx + 1, result)

def find\_subset(arr, target):

result = []

find\_subset\_util(arr, [], target, 0, result)

return result

# Example usage:

S = [1, 2, 5, 6, 8]

d = 9

subsets = find\_subset(S, d)

if subsets:

print("Subsets with sum equal to", d, ":")

for subset in subsets:

print(subset)

else:

print("No subset found with sum equal to", d)

**18. Implement N Queen's problem using back tracking.**

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

# Check if there is a queen in the same row

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on left side

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

if board[i][j] == 1:

return False

# Check lower diagonal on left side

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

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col, n, result):

if col == n:

result.append(["".join("Q" if cell == 1 else "." for cell in row) for row in board])

return True

res = False

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 1

res = solve\_n\_queens\_util(board, col + 1, n, result) or res

board[i][col] = 0 # Backtrack if placing queen at (i, col) doesn't lead to a solution

return res

def solve\_n\_queens(n):

board = [[0] \* n for \_ in range(n)]

result = []

if not solve\_n\_queens\_util(board, 0, n, result):

print("No solution exists for the given value of N.")

return []

return result

# Example usage:

n = 4

solutions = solve\_n\_queens(n)

print(f"Number of solutions for {n} queens:", len(solutions))

for i, solution in enumerate(solutions, start=1):

print(f"Solution {i}:")

for row in solution:

print(row)

**19. Write a program to implement Graph Colouring using backtracking method.**

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0] \* vertices for \_ in range(vertices)]

def is\_safe(self, v, color, c):

for i in range(self.V):

if self.graph[v][i] == 1 and color[i] == c:

return False

return True

def graph\_coloring\_util(self, m, color, v):

if v == self.V:

return True

for c in range(1, m + 1):

if self.is\_safe(v, color, c):

color[v] = c

if self.graph\_coloring\_util(m, color, v + 1):

return True

color[v] = 0

return False

def graph\_coloring(self, m):

color = [0] \* self.V

if not self.graph\_coloring\_util(m, color, 0):

print("No solution exists.")

return

print("Solution exists. The vertex colors are:")

for i in range(self.V):

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

# Example usage:

g = Graph(4)

g.graph = [

[0, 1, 1, 1],

[1, 0, 1, 0],

[1, 1, 0, 1],

[1, 0, 1, 0]

]

colors = 3 # Number of colors available

g.graph\_coloring(colors)

**20. Write a program to implement Travelling sales person using branch and bound.**

import sys

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

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

self.graph[u][v] = w

self.graph[v][u] = w

def tsp(self):

self.min\_path = sys.maxsize

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

self.visited[0] = True

self.tsp\_util(0, 1, 0, [0])

def tsp\_util(self, current, count, cost, path):

if count == self.V:

if self.graph[current][0]:

cost += self.graph[current][0]

if cost < self.min\_path:

self.min\_path = cost

self.final\_path = path + [0]

return

for i in range(self.V):

if (not self.visited[i] and

self.graph[current][i]):

self.visited[i] = True

self.tsp\_util(i, count + 1,

cost + self.graph[current][i],

path + [i])

self.visited[i] = False

# Example usage:

g = Graph(4)

g.graph = [[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]]

g.tsp()

print("Minimum cost:", g.min\_path)

print("Optimal path:", g.final\_path)

**21. Write a program to implement Travelling sales person using dynamic programming.**

import sys

def tsp(graph):

n = len(graph)

# dp array to store the minimum cost to visit each city

dp = [[-1] \* (1 << n) for \_ in range(n)]

# Function to recursively calculate the minimum cost

def dfs(node, visited):

if visited == (1 << n) - 1:

return graph[node][0] if graph[node][0] != 0 else sys.maxsize

if dp[node][visited] != -1:

return dp[node][visited]

min\_cost = sys.maxsize

for next\_node in range(n):

if visited & (1 << next\_node) == 0 and graph[node][next\_node] != 0:

cost = graph[node][next\_node] + dfs(next\_node, visited | (1 << next\_node))

min\_cost = min(min\_cost, cost)

dp[node][visited] = min\_cost

return min\_cost

return dfs(0, 1)

# Example usage:

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

min\_cost = tsp(graph)

print("Minimum cost to visit all cities:", min\_cost)

1. **Write a program to implement the backtracking algorithm for the Hamiltonian Circuits problem.**

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0] \* vertices for \_ in range(vertices)]

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

self.graph[u][v] = 1

self.graph[v][u] = 1

def is\_safe(self, v, pos, path):

# Check if vertex v is adjacent to the last vertex added to the path

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

return False

# Check if the vertex has already been visited

if v in path:

return False

return True

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

if pos == self.V:

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

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

return True

else:

return False

for v in range(1, self.V):

if self.is\_safe(v, pos, path):

path[pos] = v

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

return True

path[pos] = -1

return False

def hamiltonian\_cycle(self):

path = [-1] \* self.V

path[0] = 0

if not self.hamiltonian\_util(path, 1):

print("No Hamiltonian cycle exists.")

return False

print("Hamiltonian cycle exists. The cycle is:")

print(path)

return True

# Example usage:

g = Graph(5)

g.add\_edge(0, 1)

g.add\_edge(0, 3)

g.add\_edge(1, 2)

g.add\_edge(1, 4)

g.add\_edge(2, 3)

g.add\_edge(2, 4)

g.add\_edge(3, 4)

g.hamiltonian\_cycle()

**23. Write a program to implement greedy algorithm for job sequencing with deadlines.**

def job\_sequencing(jobs):

# Sort jobs by profit in non-decreasing order

jobs.sort(key=lambda x: x[2], reverse=True)

# Find the maximum deadline

max\_deadline = max(jobs, key=lambda x: x[1])[1]

# Initialize array to store scheduled jobs

result = [-1] \* max\_deadline

# Fill result array with scheduled jobs

total\_profit = 0

for job in jobs:

deadline = job[1]

while deadline > 0:

if result[deadline - 1] == -1:

result[deadline - 1] = job[0]

total\_profit += job[2]

break

deadline -= 1

return total\_profit, [job\_id for job\_id in result if job\_id != -1]

# Example usage:

jobs = [(1, 4, 20), (2, 1, 10), (3, 1, 40), (4, 1, 30)]

max\_profit, sequence = job\_sequencing(jobs)

print("Maximum profit:", max\_profit)

print("Job sequence:", sequence)