LAB TEST:-3

GREED TECHNIQUES

1. Single Source Shortest Paths: Dijkstra's Algorithm

Q1:Given a graph represented by an adjacency matrix, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph.The graph is represented as an adjacency matrix where graph[i][j] denote the weight of the edge from vertex i to vertex j. If there is no edge between vertices i and j, the value is Infinity (or a very large number).

Test Case 1:

Input:

n = 5

graph = [[0, 10, 3, Infinity, Infinity],[Infinity, 0, 1, 2, Infinity],[Infinity, 4, 0, 8, 2],

[Infinity, Infinity, Infinity, 0, 7],[Infinity, Infinity, Infinity, 9, 0]]

source = 0

Output:[0, 7, 3, 9, 5]

Test Case 2:

Input:

n = 4

graph = [[0, 5, Infinity, 10],[Infinity, 0, 3, Infinity],[Infinity, Infinity, 0, 1],

[Infinity, Infinity, Infinity, 0]]

source = 0

Output:[0, 5, 8, 9]

CODE:-

```python
import sys
import heapq

def dijkstra(graph, source):
    n = len(graph)
    distances = [sys.maxsize] * n
    distances[source] = 0

    pq = [(0, source)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)

        if current_distance > distances[current_vertex]:
            continue

        for adjacent_vertex, weight in enumerate(graph[current_vert
            if weight == float('inf'):
                continue

            distance = current_distance + weight

            if distance < distances[adjacent_vertex]:
                distances[adjacent_vertex] = distance
                heapq.heappush(pq, (distance, adjacent_vertex))

    return distances

# Test Case 1
n = 5
graph = [[0, 10, 3, float('inf'), float('inf')],
         [float('inf'), 0, 1, 2, float('inf')],
         [float('inf'), 4, 0, 8, 2],
         [float('inf'), float('inf'), float('inf'), 0, 7],
         [float('inf'), float('inf'), float('inf'), 9, 0]]
source = 0
print(dijkstra(graph, source))  # Output: [0, 7, 3, 9, 5]

# Test Case 2
n = 4
graph = [[0, 5, float('inf'), 10],
         [float('inf'), 0, 3, float('inf')],
         [float('inf'), float('inf'), 0, 1],
         [float('inf'), float('inf'), float('inf'), 0]]
source = 0
print(dijkstra(graph, source))  # Output: [0, 5, 8, 9]
```

OUTPUT:-

```
[0, 7, 3, 9, 5]
[0, 5, 8, 9]
```

2:Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex.The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.

Test Case 1:

Input:

n = 6

edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14),(1, 2, 10), (1, 3, 15),

(2, 3, 11), (2, 5, 2),(3, 4, 6),(4, 5, 9)]

source = 0

target = 4

Output:20

Test Case 2:

Input:

n = 5

edges = [(0, 1, 10), (0, 4, 3),(1, 2, 2), (1, 4, 4),(2, 3, 9),(3, 2, 7),(4, 1, 1), (4, 2, 8), (4, 3, 2)]

source = 0

target = 3

Output:8

CODE:-

```python
import sys
import heapq

def dijkstra(edges, n, source, target):
    graph = {i: {} for i in range(n)}
    for u, v, w in edges:
        graph[u][v] = w

    distances = {i: sys.maxsize for i in range(n)}
    distances[source] = 0

    pq = [(0, source)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)

        if current_distance > distances[current_vertex]:
            continue

        for adjacent_vertex, weight in graph[current_vertex].items()
            distance = current_distance + weight

            if distance < distances[adjacent_vertex]:
                distances[adjacent_vertex] = distance
                heapq.heappush(pq, (distance, adjacent_vertex))

    return distances[target]

# Test Case 1
n = 6
edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
         (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source = 0
target = 4
print(dijkstra(edges, n, source, target))  # Output: 20

# Test Case 2
n = 5
edges = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9),
         (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)]
source = 0
target = 3
print(dijkstra(edges, n, source, target))  # Output: 8
```

OUTPUT:-

```
26
5
```

3:Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.

Test Case 1:

Input:

n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

Output:[('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

Test Case 2:

Input:

n = 6

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

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

Output:[('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

CODE:-

```
import heapq

from collections import defaultdict


class Node:

    def __init__(self, char, freq):

        self.char = char

        self.freq = freq

        self.left = None

        self.right = None
```

```
    def __lt__(self, other):
        return self.freq < other.freq


def calculate_frequency(characters, frequencies):
    frequency_dict = {}
    for char, freq in zip(characters, frequencies):
        frequency_dict[char] = freq
    return frequency_dict


def build_heap(frequency_dict):
    heap = []
    for key in frequency_dict:
        node = Node(key, frequency_dict[key])
        heapq.heappush(heap, node)
    return heap


def merge_nodes(heap):
    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
```

```
        merged = Node(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2

        heapq.heappush(heap, merged)

def build_codes_helper(root, current_code, codes):
    if root == None:
        return

    if root.char!= None:
        codes[root.char] = current_code

    build_codes_helper(root.left, current_code + "0", codes)
    build_codes_helper(root.right, current_code + "1", codes)

def build_codes(root):
    codes = {}
    build_codes_helper(root, "", codes)
```

```
    return codes


def huffman_encoding(characters, frequencies):
    frequency_dict = calculate_frequency(characters, frequencies)
    heap = build_heap(frequency_dict)
    merge_nodes(heap)

    root = heap[0]
    codes = build_codes(root)

    encoded_chars = [(char, code) for char, code in codes.items()]
    return encoded_chars


# Test Case 1
n = 4
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
print(huffman_encoding(characters, frequencies))  # Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]
```

Test Case 2

n = 6

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

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

print(huffman_encoding(characters, frequencies)) # Output: [('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

OUTPUT:-

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]
[('a', '0'), ('d', '100'), ('c', '101'), ('f', '1100'), ('e', '1101'), ('b', '111')]
```

4:Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.

Test Case 1:

Input:

n = 4

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

encoded_string = '1101100111110'

Output:"abacd"

Test Case 2:

Input:

n = 6

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

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

encoded_string = '11001101110010111100101 1'

Output:"fcbade"

CODE:-

```
import heapq


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

    def __lt__(self, other):
        return self.freq < other.freq


def calculate_frequency(characters, frequencies):
    frequency_dict = {}
    for char, freq in zip(characters, frequencies):
```

```
        frequency_dict[char] = freq
    return frequency_dict


def build_heap(frequency_dict):
    heap = []
    for key in frequency_dict:
        node = Node(key, frequency_dict[key])
        heapq.heappush(heap, node)
    return heap


def merge_nodes(heap):
    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)

        merged = Node(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2

        heapq.heappush(heap, merged)
```

```
    return heap

def build_codes_helper(root, current_code, codes):
    if root is None:
        return

    if root.char is not None:
        codes[root.char] = current_code

    build_codes_helper(root.left, current_code + "0", codes)
    build_codes_helper(root.right, current_code + "1", codes)

def build_codes(root):
    codes = {}
    build_codes_helper(root, "", codes)
    return codes

def build_reverse_codes(codes):
    reverse_codes = {v: k for k, v in codes.items()}
    return reverse_codes
```

```python
def huffman_decoding(characters, frequencies, encoded_string):
    frequency_dict = calculate_frequency(characters, frequencies)
    heap = build_heap(frequency_dict)
    if len(heap) == 0:
        return ""

    heap = merge_nodes(heap)
    root = heap[0]
    codes = build_codes(root)
    reverse_codes = build_reverse_codes(codes)

    decoded_string = ""
    temp = ""
    for bit in encoded_string:
        temp += bit
        if temp in reverse_codes:
            decoded_string += reverse_codes[temp]
            temp = ""
```

```
    return decoded_string


# Test Case 1

characters = ['a', 'b', 'c', 'd']

frequencies = [5, 9, 12, 13]

encoded_string = '1101100111110'

print(huffman_decoding(characters, frequencies, encoded_string))  # Output: "abacd"


# Test Case 2

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

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

encoded_string = '110011011100101111001011'

print(huffman_decoding(characters, frequencies, encoded_string))  # Output: "fcbade"
```

OUTPUT:-