TOPIC 5: GREEDY

Q1. Coin Triplet Game

Aim:

To maximize coins collected following selection rules using greedy choice.

Algorithm:

1. Sort piles descending.
2. Form triplets.
3. Alice takes largest, Bob smallest, you middle.
4. Sum your selected coins.
5. Return maximum total.

Input:

```python
# Coin Triplet Game using Greedy Approach

def maxCoins(piles):
    piles.sort()
    n = len(piles)
    res = 0
    i = n // 3
    while i < n:
        res += piles[i]
        i += 2
    return res

# Input
piles = list(map(int, input("Enter coin piles separated by space: ").split()))
print("Maximum coins you can collect:", maxCoins(piles))

```

Output:

```
PS C:\Users\volap> & C:/Users/volap/AppData/Local/Program
Enter coin piles separated by space: 2 4 1 2 7 8
Maximum coins you can collect: 9
```

Result: The program correctly selects coins using the greedy strategy to maximise the total value. It outputs the highest possible sum of coins collected.

Q2. Dijkstra's Algorithm (Matrix Form)

Aim:

To find shortest distance from a source to all vertices using Dijkstra's algorithm.

Algorithm:

1. Read adjacency matrix.
2. Initialize distances = ∞.
3. Set distance[source]=0.
4. Update distances using relaxation.
5. Return final shortest path list.

Input:

```python
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    while pq:
        curr_dist, curr_node = heapq.heappop(pq)
        if curr_dist > distances[curr_node]:
            continue
        for neighbor, weight in graph[curr_node].items():
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances

# Input
graph = {
    'A': {'B': 4, 'C': 1},
    'B': {'A': 4, 'C': 2, 'D': 5},
    'C': {'A': 1, 'B': 2, 'D': 8},
    'D': {'B': 5, 'C': 8}
}
start = input("Enter start node: ")
result = dijkstra(graph, start)
print("Shortest distances from", start, ":", result)
```

Output:

```
PS C:\Users\volap> & C:/Users/volap/AppData/Local/Programs/Pyt
Enter start node: A
Shortest distances from A : {'A': 0, 'B': 3, 'C': 1, 'D': 8}
```

Result: The shortest path from the source node to all others is computed efficiently. The greedy edge selection ensures minimum total distance.

Q3. Huffman Coding

Aim:

To generate optimal prefix codes using Huffman tree.

Algorithm:

1. Create a min-heap of character frequencies.
2. Extract two smallest nodes.
3. Merge them into a new internal node.
4. Repeat until one node left.
5. Assign binary codes.

Input:

```python
import heapq

def huffman_code(symbols, freq):
    heap = [[f, [s, ""]] for s, f in zip(symbols, freq)]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

# Input
symbols = list(input("Enter symbols separated by space: ").split())
freq = list(map(int, input("Enter frequencies separated by space: ").split()))
huff = huffman_code(symbols, freq)
print("Symbol\tFrequency\tHuffman Code")
for s, code in huff:
    print(f"{s}\t\t{freq[symbols.index(s)]}\t\t{code}")
```

Output:

```
PS C:\Users\volap> & C:/Users/volap/AppData/Local/P
Enter symbols separated by space: a b c d
Enter frequencies separated by space: 5 9 12 13
Symbol  Frequency       Huffman Code
a               5               00
b               9               01
c               12              10
d               13              11
```

Result: The algorithm generates optimal binary prefix codes for each symbol. It reduces average code length, achieving data compression.

Q4. Kruskal's MST Algorithm

Aim:
To find a minimum spanning tree of a connected graph.
Algorithm:

1. Sort edges by weight.
2. Initialize disjoint sets.
3. Add edge if no cycle formed.
4. Repeat until n – 1 edges.
5. Output MST and total cost.

Input:

```
def KruskalMST(self):
    result = []
    i = e = 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 += 1
        x = self.find(parent, u)
        y = self.find(parent, v)
        if x != y:
            e += 1
            result.append([u, v, w])
            self.union(parent, rank, x, y)
    print("Edges in MST:")
    for u, v, w in result:
        print(f"{u} -- {v} == {w}")
```

Output:

```
PS C:\Users\volap> & C:/Users
Edges in MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Result: The program constructs a minimum-cost spanning tree using edge-weight sorting.It ensures all vertices are connected with minimal total cost.

Q5. Fractional Knapsack

Aim:
To maximize profit by selecting items partially using greedy ratio sorting.
Algorithm:

1. Read weights & values.
2. Compute value/weight ratio.
3. Sort descending by ratio.
4. Pick items until full capacity.
5. Add fraction if partial fit.

Input:

```python
# Fractional Knapsack Problem

def fractionalKnapsack(W, val, wt):
    ratio = [[v / w, v, w] for v, w in zip(val, wt)]
    ratio.sort(reverse=True)
    total_value = 0
    for r, v, w in ratio:
        if W >= w:
            W -= w
            total_value += v
        else:
            total_value += r * W
            break
    return total_value

# Input
W = int(input("Enter capacity of knapsack: "))
val = list(map(int, input("Enter values: ").split()))
wt = list(map(int, input("Enter weights: ").split()))

print("Maximum value in knapsack:", fractionalKnapsack(W, val, wt))

```

Output:

```
PS C:\Users\volap> & C:/Users/volap/Ap
Enter capacity of knapsack: 50
Enter values: 60 100 120
Enter weights: 10 20 30
Maximum value in knapsack: 240.0
```

Result: The greedy selection by value-to-weight ratio maximises total profit.
It achieves the optimal fractional solution within given capacity.