GREEDY: PMST

```
import heapq
from collections import defaultdict

# Graph class to represent an undirected weighted graph
class Graph:
    def __init__(self):
        # Using defaultdict to store adjacency list: {vertex: [(neighbor, weight), ...]}
        self.graph = defaultdict(list)

    def add_edge(self, u, v, weight):
        """
        Add edge to the graph (undirected).
        That means both directions: u -> v and v -> u
        """
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))

    def prim_mst(self, start_vertex):
        """
        Implements Prim's algorithm to find the Minimum Spanning Tree (MST)
        using a priority queue (min-heap).
        """
        visited = set()                # To track visited vertices
        min_heap = [(0, start_vertex, None)] # (weight, current_vertex, parent_vertex)
        mst_cost = 0                   # Total cost of MST
        mst_edges = []                 # List to store MST edges
```

```
        while min_heap:
            weight, current_vertex, parent = heapq.heappop(min_heap)

            # Skip if the vertex is already visited
            if current_vertex in visited:
                continue

            visited.add(current_vertex)
            mst_cost += weight

            # Add edge to MST result (skip the first 0-weight edge)
            if parent is not None:
                mst_edges.append((parent, current_vertex, weight))

            # Traverse all adjacent vertices
            for neighbor, edge_weight in self.graph[current_vertex]:
                if neighbor not in visited:
                    # Add to priority queue with current_vertex as parent
                    heapq.heappush(min_heap, (edge_weight, neighbor, current_vertex))

        # Print the edges in the MST
        print("Edges in the Minimum Spanning Tree:")
        for u, v, w in mst_edges:
            print(f"{u} - {v} with weight {w}")
        print("Total cost of MST:", mst_cost)

# Example usage of the Graph and Prim's Algorithm
if __name__ == "__main__":
```

```python
g = Graph()

# Add edges: g.add_edge(u, v, weight)

g.add_edge('A', 'B', 2)

g.add_edge('A', 'C', 3)

g.add_edge('B', 'C', 1)

g.add_edge('B', 'D', 4)

g.add_edge('C', 'D', 5)

g.add_edge('C', 'E', 6)

g.add_edge('D', 'E', 7)


print("Prim's MST starting from vertex A:")

g.prim_mst('A')
```

```
Prim's MST starting from vertex A:
Edges in the Minimum Spanning Tree:
A - B with weight 2
B - C with weight 1
B - D with weight 4
C - E with weight 6
Total cost of MST: 13
```

Viva Questions for This Practical

◆ **1. What is Prim's Algorithm?**

Answer:
Prim’s Algorithm is used to find the **Minimum Spanning Tree (MST)** of a connected, weighted graph. It connects all vertices with the **minimum total edge weight** and **no cycles**.

◆ **2. What is a Minimum Spanning Tree (MST)?**

Answer:
A Minimum Spanning Tree is a subset of edges that connects all the vertices of a graph with:

- The **least total weight**,

- **No cycles**, and
- **All nodes included**.

◆ **3. What is the use of Prim's Algorithm?**

Answer:
It is used in:

- Network design (e.g., telephone or computer networks),
- Road construction planning,
- Electrical grids,
- Clustering in AI/ML.

◆ **4. How does Prim's Algorithm work in simple terms?**

Answer:
Start from any one vertex and:

- Add the **cheapest edge** connecting to a **new vertex**.
- Repeat this until all vertices are connected and **no cycles** are formed.

◆ **5. What data structures are used in your Python code?**

Answer:

- defaultdict(list) – for the graph's adjacency list.
- heapq – for selecting the **smallest edge** quickly (priority queue).
- set – to keep track of visited vertices.
- list – to store final MST edges.

◆ **6. Why do we use a priority queue (heap)?**

Answer:
A priority queue helps us always pick the **edge with the smallest weight**, which is necessary for the MST to be minimum.

🔸 7. Can Prim’s Algorithm work on disconnected graphs?

Answer:
No, it only works on **connected** graphs. If the graph is disconnected, it cannot create an MST.

🔸 8. What is the time complexity of Prim’s Algorithm?

Answer:

- With a **min-heap and adjacency list**, time complexity is **O(E log V)**, where E is edges and V is vertices.

🔸 9. What’s the difference between Prim’s and Kruskal’s Algorithm?

Feature	Prim's Algorithm	Kruskal's Algorithm
Approach	Vertex-based	Edge-based
Data structure	Heap, Adjacency List	Disjoint Set (Union-Find)
Graph type	Works better with dense graph	Works better with sparse graph

---

🔸 10. What happens if there are two edges with the same weight?

Answer:
The algorithm can choose either edge, as both are valid. The final MST may look different but will have the **same total weight**.

🔸 11. How do you avoid cycles in Prim's Algorithm?

Answer:
We avoid cycles by **not revisiting already visited nodes** using a visited set.

🔸 12. What is the role of prev_vertex or parent in the code?

Answer:
It helps us **track from which node we came** so we can print or store the edge correctly in the MST result.

◆ 13. What if all edge weights are the same?

Answer:
The MST can still be formed. There may be **multiple valid MSTs**, all with the same total cost.

◆ 14. Can MST have more than one solution?

Answer:
Yes, if the graph has **multiple edges with the same weight**, there can be **multiple MSTs** with the **same total cost**.

◆ 15. What if the graph has negative weights?

Answer:
Prim’s Algorithm **can handle negative weights** as long as the graph is connected and has **no negative cycles**.
