Min Cost to collect all cities in C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Edge {
   int v;
   int wt;

    Edge(int nbr, int weight) {
      this->v = nbr;
      this->wt = weight;
   }
};

struct CompareEdge {
   bool operator()(const Edge& e1, const Edge& e2) {
      return e1.wt > e2.wt; // Min-Heap based on edge
weight
   }
};

int main() {
   // Hardcoded input
   int vtces = 7;
   int edges = 8;
   vector<vector<Edge>> graph(vtces);

   // Hardcoded edges
   vector<vector<int>> hardcoded_edges = {
      {0, 1, 10},
      {1, 2, 10},
      {2, 3, 10},
      {0, 3, 40},
      {3, 4, 2},
      {4, 5, 3},
      {5, 6, 3},
      {4, 6, 8}
   };

   // Populating the graph with hardcoded edges
   for (auto& edge : hardcoded_edges) {
      int v1 = edge[0];
      int v2 = edge[1];
      int wt = edge[2];
      graph[v1].emplace_back(v2, wt);
      graph[v2].emplace_back(v1, wt);
   }

   int ans = 0;
   priority_queue<Edge, vector<Edge>, CompareEdge>
pq;
   vector<bool> vis(vtces, false);
   pq.push(Edge(0, 0)); // Start with any vertex (0 in this
case) with 0 weight

   while (!pq.empty()) {
      Edge rem = pq.top();
      pq.pop();

      if (vis[rem.v]) {
```

Step-by-Step Execution

Step 1: Populate the Graph

The adjacency list (graph) for the given input will look like this:

0: (1, 10), (3, 40)
1: (0, 10), (2, 10)
2: (1, 10), (3, 10)
3: (2, 10), (0, 40), (4, 2)
4: (3, 2), (5, 3), (6, 8)
5: (4, 3), (6, 3)
6: (5, 3), (4, 8)

Step 2: Initialize Priority Queue and Visited Array

- Start with vertex 0 and push an edge (0, 0) to the priority queue (pq).
- Initially:

  pq: [(0, 0)] (min-heap: weight 0, vertex 0)
  vis: [false, false, false, false, false, false, false]
  ans: 0

Step 3: Prim's Algorithm

- **Iteration 1:**
  - Pop (0, 0) from pq.
  - Add weight 0 to ans. Now ans = 0.
  - Mark vertex 0 as visited (vis[0] = true).
  - Push neighbors (1, 10) and (3, 40) to pq.

  pq: [(1, 10), (3, 40)]
  vis: [true, false, false, false, false, false, false]
  ans: 0

- **Iteration 2:**
  - Pop (1, 10) from pq.
  - Add weight 10 to ans. Now ans = 10.
  - Mark vertex 1 as visited (vis[1] = true).
  - Push neighbors (2, 10) to pq (skip (0, 10) because 0 is already visited).

  pq: [(2, 10), (3, 40)]
  vis: [true, true, false, false, false, false, false]

```
            continue;
        }
        vis[rem.v] = true;
        ans += rem.wt;

        for (Edge nbr : graph[rem.v]) {
            if (!vis[nbr.v]) {
                pq.push(nbr);
            }
        }
    }

    cout << ans << endl;

    return 0;
}
```

ans: 10

- **Iteration 3:**
    - Pop (2, 10) from pq.
    - Add weight 10 to ans. Now ans = 20.
    - Mark vertex 2 as visited (vis[2] = true).
    - Push neighbor (3, 10) to pq (skip (1, 10) because 1 is already visited).

  pq: [(3, 10), (3, 40)]
  vis: [true, true, true, false, false, false, false]
  ans: 20

- **Iteration 4:**
    - Pop (3, 10) from pq.
    - Add weight 10 to ans. Now ans = 30.
    - Mark vertex 3 as visited (vis[3] = true).
    - Push neighbors (4, 2) to pq (skip (2, 10) and (0, 40) because 2 and 0 are already visited).

  pq: [(3, 40), (4, 2)]
  vis: [true, true, true, true, false, false, false]
  ans: 30

- **Iteration 5:**
    - Pop (4, 2) from pq.
    - Add weight 2 to ans. Now ans = 32.
    - Mark vertex 4 as visited (vis[4] = true).
    - Push neighbors (5, 3) and (6, 8) to pq (skip (3, 2) because 3 is already visited).

  pq: [(3, 40), (5, 3), (6, 8)]
  vis: [true, true, true, true, true, false, false]
  ans: 32

- **Iteration 6:**
    - Pop (5, 3) from pq.
    - Add weight 3 to ans. Now ans = 35.
    - Mark vertex 5 as visited (vis[5] = true).
    - Push neighbor (6, 3) to pq (skip (4, 3) because 4 is already visited).

  pq: [(3, 6), (6, 8), (3, 40)]
  vis: [true, true, true, true, true, true, false]

	ans: 35
	
	• **Iteration 7:**
	    ○ Pop (6, 3) from pq.
	    ○ Add weight 3 to ans. Now ans = 38.
	    ○ Mark vertex 6 as visited (vis[6] = true).
	    ○ Skip pushing neighbors because all are already visited.
	
	pq: [(3, 40), (6, 8)]
	vis: [true, true, true, true, true, true, true]
	ans: 38
	
	Final MST Weight:
	
	• All vertices are visited, and the MST weight is 38.

Output:- 38	