```cpp
//Algorithm

#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <climits>

using namespace std;

// Define the structure for an edge in the graph
struct Edge {
    int to, weight;
};

// Function to perform Dijkstra's algorithm
void dijkstra(int start, const vector<vector<Edge>>& graph, vector<int>&
distances) {
    // Initialize the distance to the start node as 0
    distances[start] = 0;

    // Priority queue to store (distance, node) pairs, ordered by distance
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        int current_distance = pq.top().first;
        int current_node = pq.top().second;
        pq.pop();

        // If the current distance is greater than the recorded distance,
skip this node
        if (current_distance > distances[current_node]) {
            continue;
        }

        // Iterate over all adjacent nodes
        for (const Edge& edge : graph[current_node]) {
            int next_node = edge.to;
            int weight = edge.weight;
            int distance_through_current = current_distance + weight;

            // If a shorter path to the next node is found, update the
distance and add to the queue
            if (distance_through_current < distances[next_node]) {
                distances[next_node] = distance_through_current;
                pq.push({distance_through_current, next_node});
            }
        }
    }
}
int main() {
    // Number of exhibits
    int numExhibits = 4;
    // Define the graph with nodes and weighted edges
    vector<vector<Edge>> graph(numExhibits);
```

```
    // Define edges (paths between exhibits with distances)
    graph[0].push_back({1, 10}); // Z-B
    graph[0].push_back({2, 15}); // Z-C
    graph[1].push_back({2, 5});  // B-C
    graph[1].push_back({3, 10}); // B-X
    graph[2].push_back({3, 20}); // C-X
    // Visitor's rankings of exhibits (lower rank means higher preference)
    unordered_map<int, int> preferences = {
            {0, 1}, // Z (Entrance)
            {1, 3}, // B
            {2, 2}, // C
            {3, 4}  // X (Exit)
    };
    // Multiplier for the ranking system
    int multiplier = 5;

    // Adjust weights based on preferences
    for (int i = 0; i < numExhibits; ++i) {
        for (Edge& edge : graph[i]) {
            edge.weight += preferences[edge.to] * multiplier;
        }
    }
    // Vector to store the minimum distances from the start node
    vector<int> distances(numExhibits, INT_MAX);
    // Run Dijkstra's algorithm from node 0 (Entrance)
    dijkstra(0, graph, distances);
    // Output the shortest path to node 3 (Exit)
    cout << "Shortest path from Entrance (Z) to Exit (X) has a weight of: "
<< distances[3] << endl;
    return 0;
}
```