Assignment no.2

Input-

```
#include <iostream>

#include <vector> #include

<queue>

#include <map>

#include <limits>


using namespace std;


// Structure to represent a directed edge in the graph struct

Edge {

   int target;

int weight;

   Edge(int tgt, int w) : target(tgt), weight(w) {}

};


// Structure to represent a node in the graph struct

Node {

   int id;     int

heuristic;

   Node(int i, int h) : id(i), heuristic(h) {}

};


// Comparator for priority queue based on total cost

struct CompareNode {     bool operator()(const Node& a,

const Node& b) const {

       return a.heuristic > b.heuristic;

   }
```

```
};

// A* search algorithm
void astar(const vector<vector<Edge> >& graph, const vector<int>& heuristic, int source, int
target) {     int n = graph.size();

   // Priority queue to store nodes based on heuristic value
priority_queue<Node, vector<Node>, CompareNode> pq;

   // Map to store the best known cost to reach each node
map<int, int> best_cost;

   // Map to store the parent node of each node in the shortest path
map<int, int> parent;

   // Initialize the priority queue with the source node
pq.push(Node(source, heuristic[source]));

   // Initialize the best cost to reach all nodes to infinity
   for (int i = 0; i < n; ++i) {
      best_cost[i] = numeric_limits<int>::max();
   }

   // Cost to reach the source node is 0
best_cost[source] = 0;

   while (!pq.empty()) {
Node current = pq.top();
      pq.pop();
```

```
        // If target node is reached, break
if (current.id == target) {
            break;
        }

        // Explore neighbors of the current node
        for (int j = 0; j < graph[current.id].size(); ++j) {
    const Edge& edge = graph[current.id][j];     int
neighbor = edge.target;     int new_cost =
best_cost[current.id] + edge.weight;

    // If a shorter path to neighbor is found     if (new_cost <
best_cost[neighbor]) {        best_cost[neighbor] = new_cost;
parent[neighbor] = current.id;        pq.push(Node(neighbor,
new_cost + heuristic[neighbor]));
    }
   }
    }

    // Output the shortest path
vector<int> path;     int
current = target;     while
(current != source) {
path.push_back(current);
current = parent[current];
    }
    path.push_back(source);
```

```
    cout << "Shortest Path: ";
    for (int i = path.size() - 1; i >= 0; --i) {
cout << path[i];
        if (i != 0) {
cout << " -> ";
        }
    }
    cout << endl;
}

int main() {
    // Input the number of nodes and edges
    int n, m;
    cout << "Enter the number of nodes and edges: ";
    cin >> n >> m;

    // Initialize the graph
    vector<vector<Edge> > graph(n);

    // Input the directed edges and their weights     cout << "Enter the edges
and their weights (source target weight):" << endl;
    for (int i = 0; i < m; ++i) {         int source, target,
weight;         cin >> source >> target >> weight;
graph[source].push_back(Edge(target, weight));
    }

    // Input the heuristic values for each node
    vector<int> heuristic(n);     cout << "Enter the heuristic
values for each node:" << endl;
```

```
   for (int i = 0; i < n; ++i) {
cin >> heuristic[i];
   }

   // Input the source and target nodes
   int source, target;    cout << "Enter the source
and target nodes: ";    cin >> source >> target;

   // Perform A* search    astar(graph,
heuristic, source, target);

   return 0;
}
```

Output-

Enter the number of nodes and edges: 5 7

Enter the edges and their weights (source target weight):

0 1 1

0 2 4

1 2 2

1 4 5

1 3 12

2 4 2

4 3 3

Enter the heuristic values for each node:

7 6 2 0 1

Enter the source and target nodes: 0 3

Shortest Path: 0 -> 1 -> 2 -> 4 -> 3