As an assignment, students at HackerLand High School are to find a subsequence using two strings by performing the below-mentioned operation. Given two strings *firstString* of length *n* and *secondString* of length *m*, the goal is to make *secondString* a subsequence of *firstString* by applying the operation any number of times.

In one operation, any single character can be removed from the *secondString*. The goal is to find the minimum possible *difference value* which is calculated as:

| maximum index of all the characters removed from the string *secondString* | - |minimum index of all the characters removed from the string *secondString* | + 1. Removing a character from *secondString* does not affect the indices of the other characters and an empty string is always a subsequence of *firstString*.

**Note:** A subsequence of a string is a new string formed deleting some (can be none) of the characters from a string without changing the relative positions of the remaining characters. "ace" is a subsequence of "abcde" but "aec" is not.

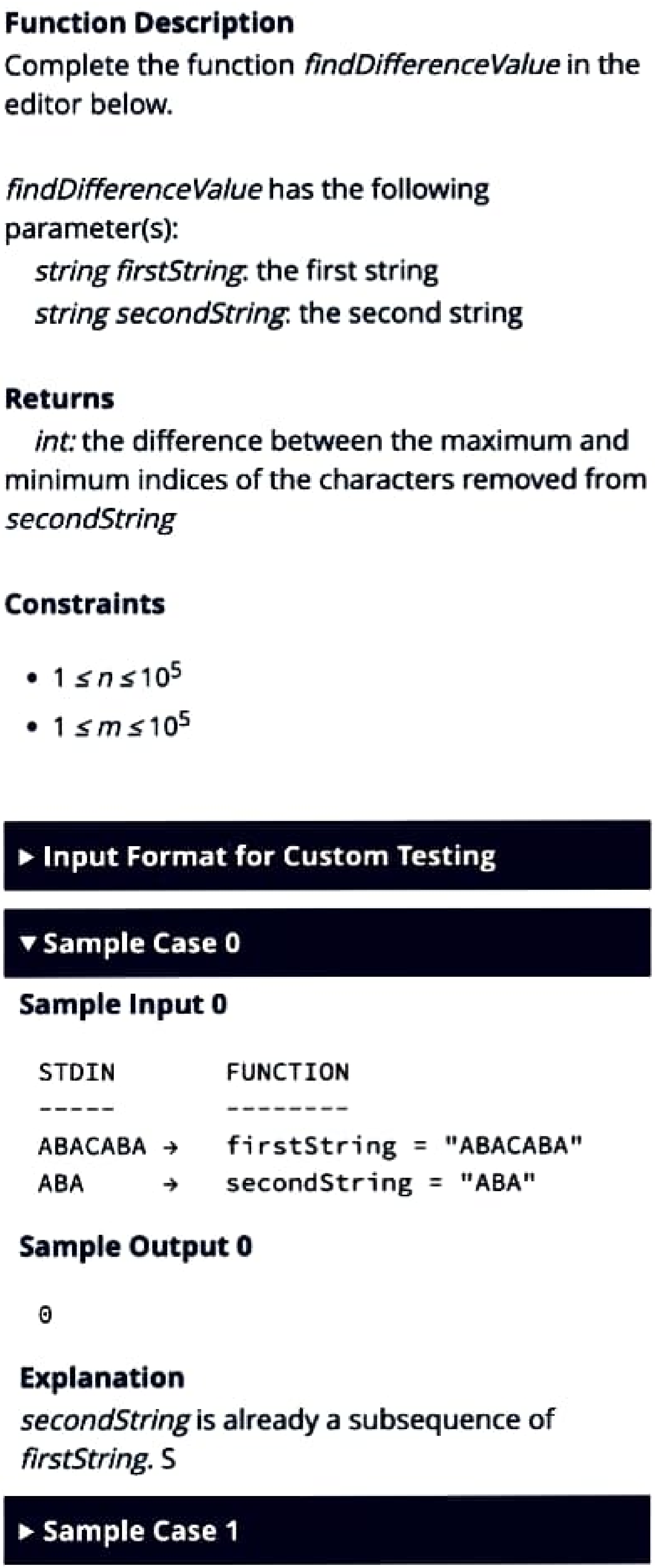
**Example**

*n = 10, firstString = HACKERRANK*

*m = 9, secondString = HACKERMAN*

Remove the character at index 7 to change *secondString* to *"HACKERAN"*, a subsequence of *firstString*. The difference value is 7 - 7 + 1 = 1. Return 1.

**Function Description**



**Function Description**

Complete the function *findDifferenceValue* in the editor below.

*findDifferenceValue* has the following parameter(s):

*string firstString*: the first string
*string secondString*: the second string

**Returns**

*int:* the difference between the maximum and minimum indices of the characters removed from *secondString*

**Constraints**

- $1 \le n \le 10^5$
- $1 \le m \le 10^5$

▶ **Input Format for Custom Testing**

▼ **Sample Case 0**

**Sample Input 0**

```
STDIN        FUNCTION
-----        --------
ABACABA →    firstString = "ABACABA"
ABA     →    secondString = "ABA"
```

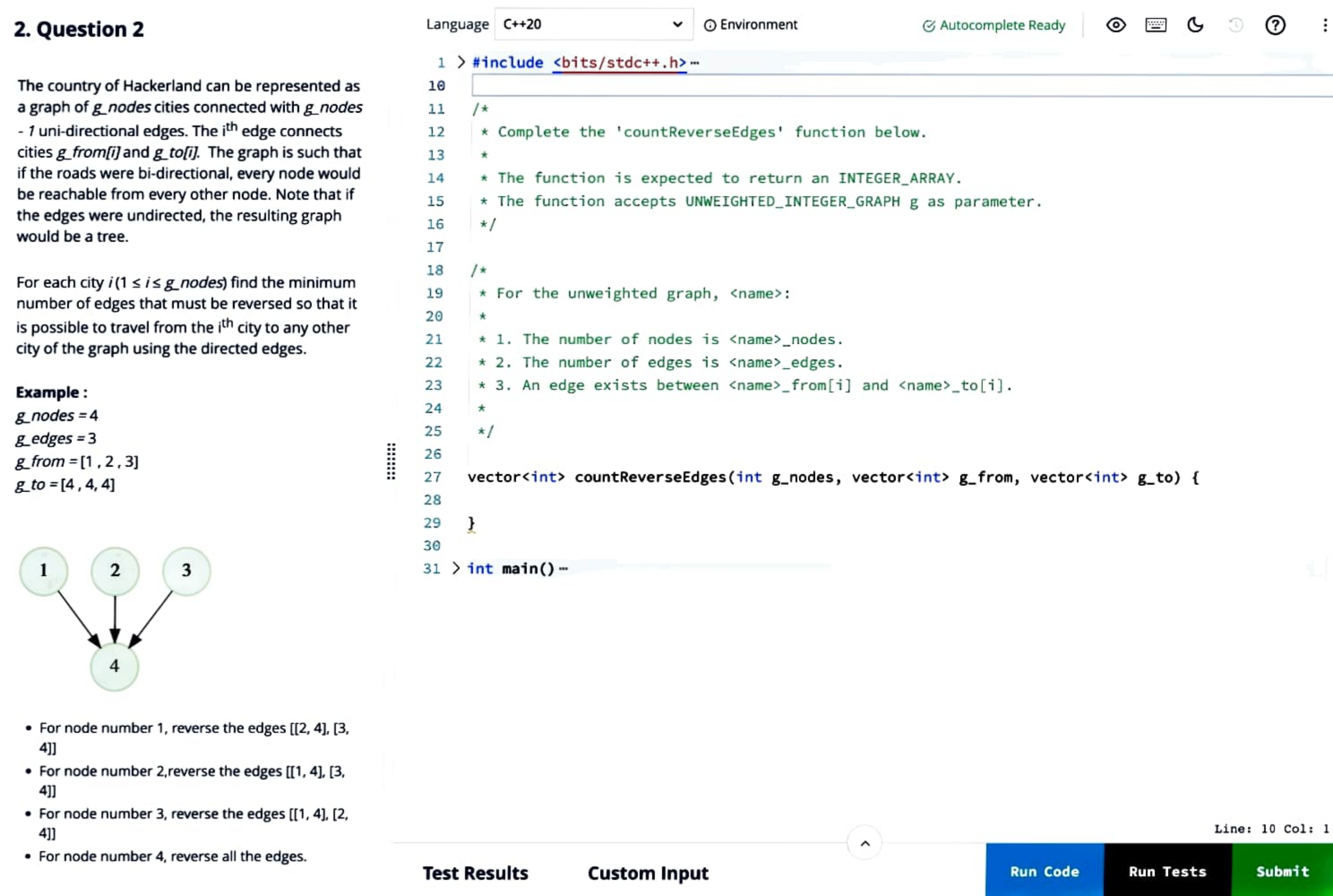
**Sample Output 0**

```
0
```

**Explanation**

*secondString* is already a subsequence of *firstString*. S

▶ **Sample Case 1**



## 2. Question 2

The country of Hackerland can be represented as a graph of *g_nodes* cities connected with *g_nodes - 1* uni-directional edges. The i th edge connects cities *g_from[i]* and *g_to[i]*. The graph is such that if the roads were bi-directional, every node would be reachable from every other node. Note that if the edges were undirected, the resulting graph would be a tree.

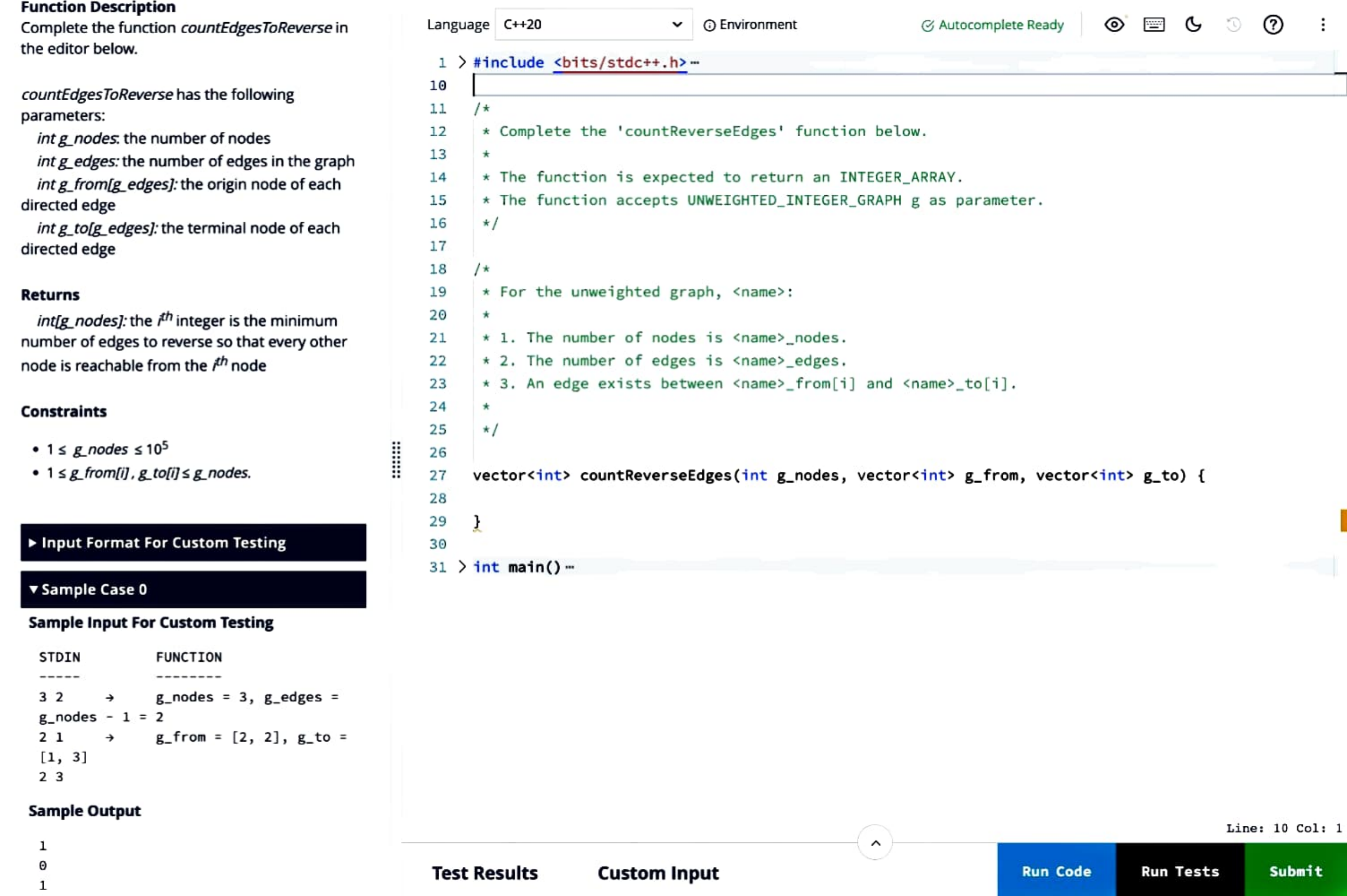
For each city *i* (1 ≤ *i* ≤ *g_nodes*) find the minimum number of edges that must be reversed so that it is possible to travel from the i th city to any other city of the graph using the directed edges.

**Example :**

*g_nodes* = 4
*g_edges* = 3
*g_from* = [1 , 2 , 3]
*g_to* = [4 , 4, 4]

- For node number 1, reverse the edges [[2, 4], [3, 4]]
- For node number 2,reverse the edges [[1, 4], [3, 4]]
- For node number 3, reverse the edges [[1, 4], [2, 4]]
- For node number 4, reverse all the edges.

```cpp
#include <bits/stdc++.h>…

/*
 * Complete the 'countReverseEdges' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts UNWEIGHTED_INTEGER_GRAPH g as parameter.
 */

/*
 * For the unweighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i].
 *
 */

vector<int> countReverseEdges(int g_nodes, vector<int> g_from, vector<int> g_to) {

}

int main()…
```



### Function Description

Complete the function *countEdgesToReverse* in the editor below.

*countEdgesToReverse* has the following parameters:

*int g_nodes*: the number of nodes

*int g_edges:* the number of edges in the graph

*int g_from[g_edges]:* the origin node of each directed edge

*int g_to[g_edges]:* the terminal node of each directed edge

### Returns

*int[g_nodes]:* the $i^{th}$ integer is the minimum number of edges to reverse so that every other node is reachable from the $i^{th}$ node

### Constraints

- $1 \le g\_nodes \le 10^5$
- $1 \le g\_from[i], g\_to[i] \le g\_nodes.$

### Sample Input For Custom Testing

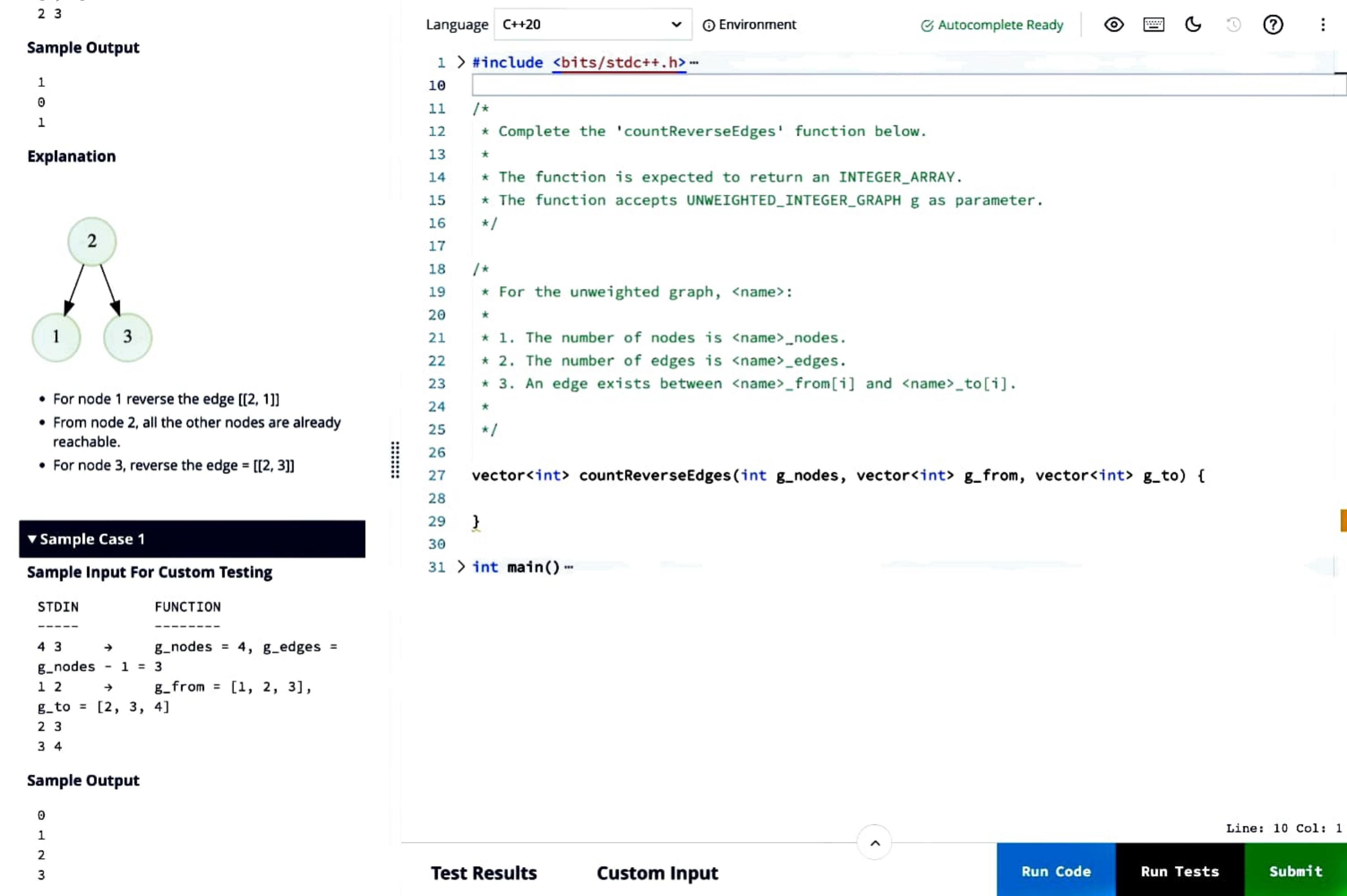
```
STDIN         FUNCTION
-----         --------
3 2       →   g_nodes = 3, g_edges =
g_nodes - 1 = 2
2 1       →   g_from = [2, 2], g_to =
[1, 3]
2 3
```

### Sample Output

```
1
0
1
```

```
#include <bits/stdc++.h>…

/*
 * Complete the 'countReverseEdges' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts UNWEIGHTED_INTEGER_GRAPH g as parameter.
 */

/*
 * For the unweighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i].
 *
 */

vector<int> countReverseEdges(int g_nodes, vector<int> g_from, vector<int> g_to) {

}

int main()…
```

Line: 10 Col: 1



```
2 3
```

**Sample Output**

```
1
0
1
```

**Explanation**

- For node 1 reverse the edge [[2, 1]]
- From node 2, all the other nodes are already reachable.
- For node 3, reverse the edge = [[2, 3]]

**▼ Sample Case 1**

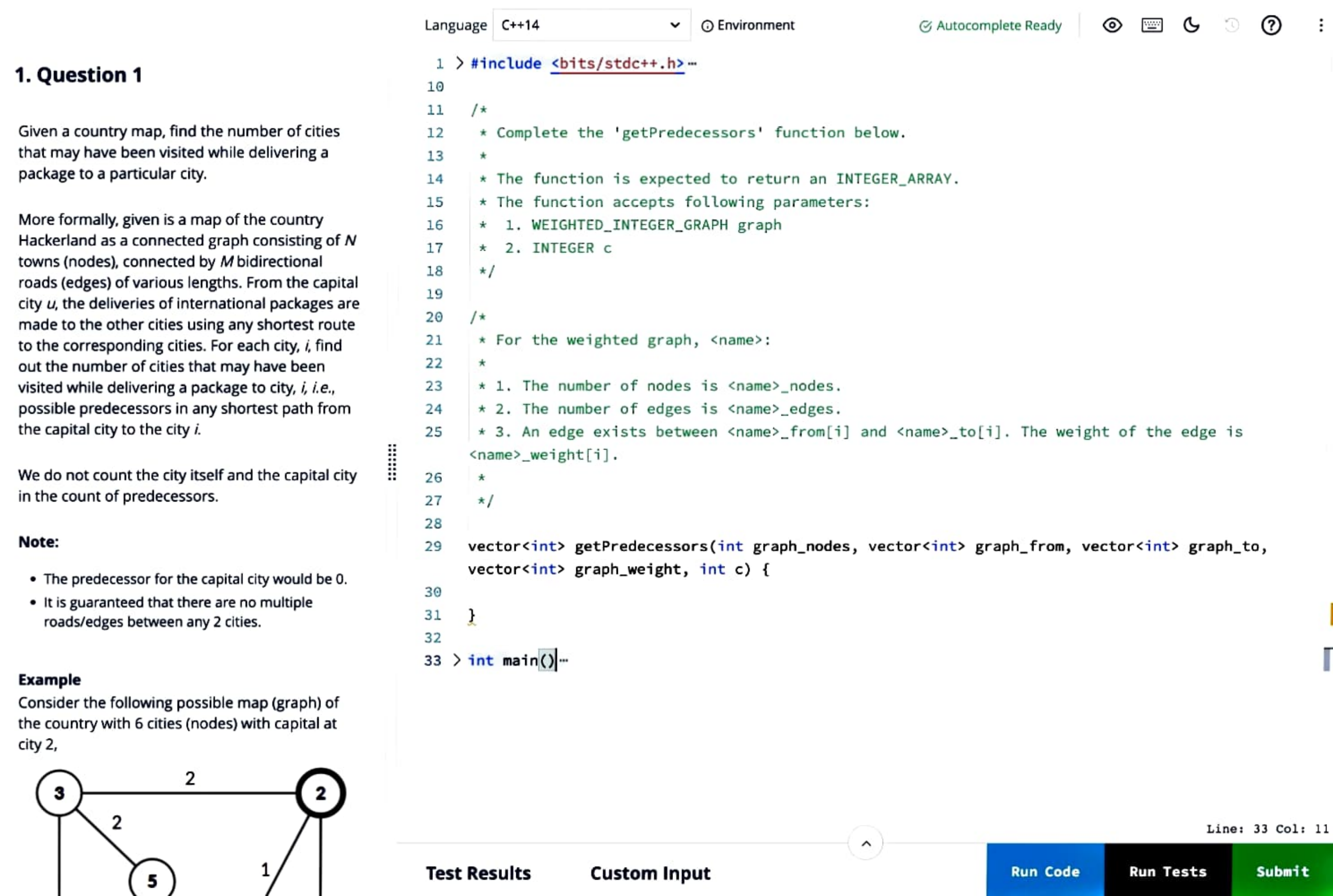
**Sample Input For Custom Testing**

```
STDIN         FUNCTION
-----         --------
4 3      →    g_nodes = 4, g_edges =
g_nodes - 1 = 3
1 2      →    g_from = [1, 2, 3],
g_to = [2, 3, 4]
2 3
3 4
```

**Sample Output**

```
0
1
2
3
```

```cpp
#include <bits/stdc++.h>...

/*
 * Complete the 'countReverseEdges' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts UNWEIGHTED_INTEGER_GRAPH g as parameter.
 */

/*
 * For the unweighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i].
 *
 */

vector<int> countReverseEdges(int g_nodes, vector<int> g_from, vector<int> g_to) {

}

int main()...
```



## 1. Question 1

Given a country map, find the number of cities that may have been visited while delivering a package to a particular city.

More formally, given is a map of the country Hackerland as a connected graph consisting of *N* towns (nodes), connected by *M* bidirectional roads (edges) of various lengths. From the capital city *u*, the deliveries of international packages are made to the other cities using any shortest route to the corresponding cities. For each city, *i*, find out the number of cities that may have been visited while delivering a package to city, *i*, *i.e.*, possible predecessors in any shortest path from the capital city to the city *i*.

We do not count the city itself and the capital city in the count of predecessors.

**Note:**

- The predecessor for the capital city would be 0.
- It is guaranteed that there are no multiple roads/edges between any 2 cities.

**Example**

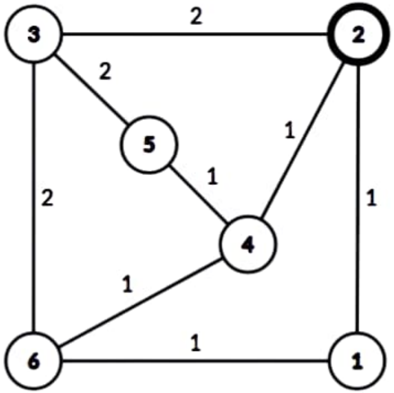
Consider the following possible map (graph) of the country with 6 cities (nodes) with capital at city 2,

```
#include <bits/stdc++.h>⋯

/*
 * Complete the 'getPredecessors' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts following parameters:
 *  1. WEIGHTED_INTEGER_GRAPH graph
 *  2. INTEGER c
 */

/*
 * For the weighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
 *
 */

vector<int> getPredecessors(int graph_nodes, vector<int> graph_from, vector<int> graph_to, vector<int> graph_weight, int c) {

}

int main()⋯
```

Line: 33 Col: 11

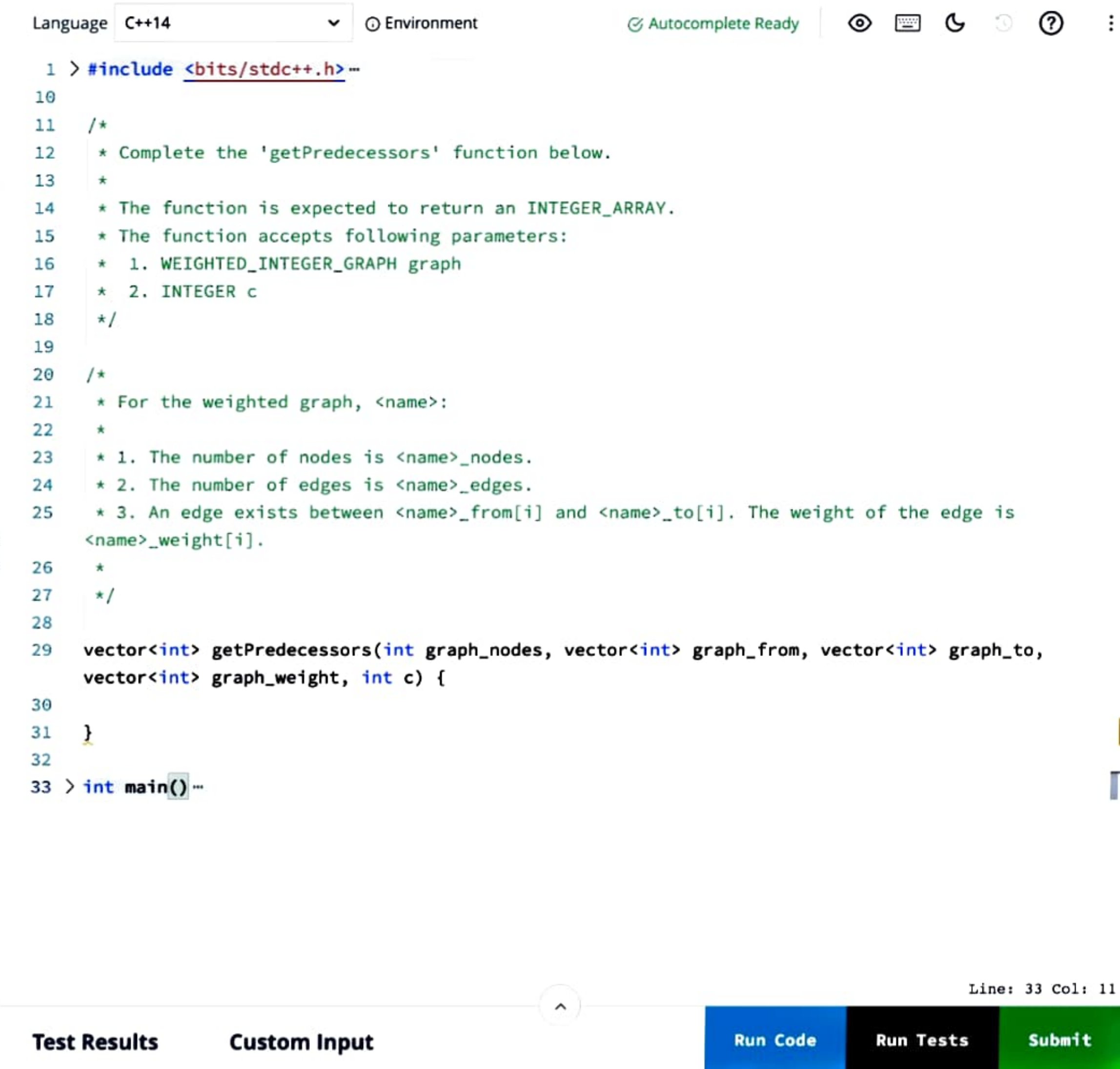
**Example**

Consider the following possible map (graph) of the country with 6 cities (nodes) with capital at city 2,

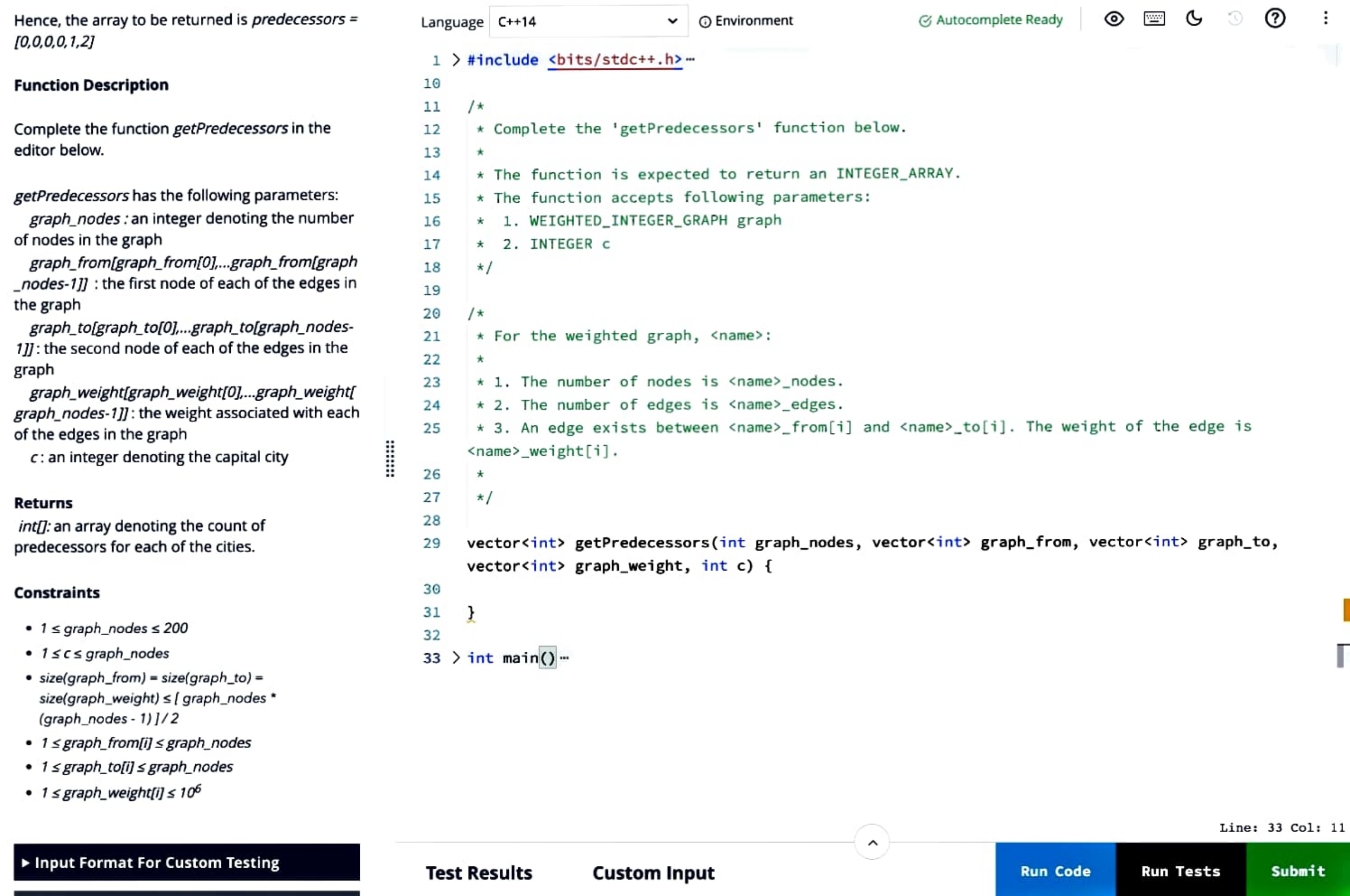


Let us look at the shortest paths possible to each of the cities from the capital city 2 :

- City 1: 2 → 1 shortest path of length 1 directly through edge from the capital city, so no predecessor for 1.
- City 2: With 2 being the capital city, naturally, it has no predecessor.
- City 3: 2 →3 shortest path of length 2 directly through edge from the capital city, so no predecessor for 3.
- City 4: 2 →4 shortest path of length 1 directly through edge from the capital city, so no predecessor for 4.
- City 5: 2 →4 →5, shortest path length of 2 through intermediate city 4, there is no other path from 2 shorter than this, hence 1 predecessor for 5.
- City 6: 2 shortest paths to city 6, 2 →4 →6 and 2 →1 →6, hence 2 predecessors for 6.



```cpp
#include <bits/stdc++.h>…

/*
 * Complete the 'getPredecessors' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts following parameters:
 *  1. WEIGHTED_INTEGER_GRAPH graph
 *  2. INTEGER c
 */

/*
 * For the weighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
 *
 */

vector<int> getPredecessors(int graph_nodes, vector<int> graph_from, vector<int> graph_to, vector<int> graph_weight, int c) {

}

int main()…
```



Hence, the array to be returned is *predecessors = [0,0,0,0,1,2]*

**Function Description**

Complete the function *getPredecessors* in the editor below.

*getPredecessors* has the following parameters:

*graph_nodes :* an integer denoting the number of nodes in the graph

*graph_from[graph_from[0],...graph_from[graph_nodes-1]]* : the first node of each of the edges in the graph

*graph_to[graph_to[0],...graph_to[graph_nodes-1]]* : the second node of each of the edges in the graph

*graph_weight[graph_weight[0],...graph_weight[graph_nodes-1]]* : the weight associated with each of the edges in the graph

*c* : an integer denoting the capital city

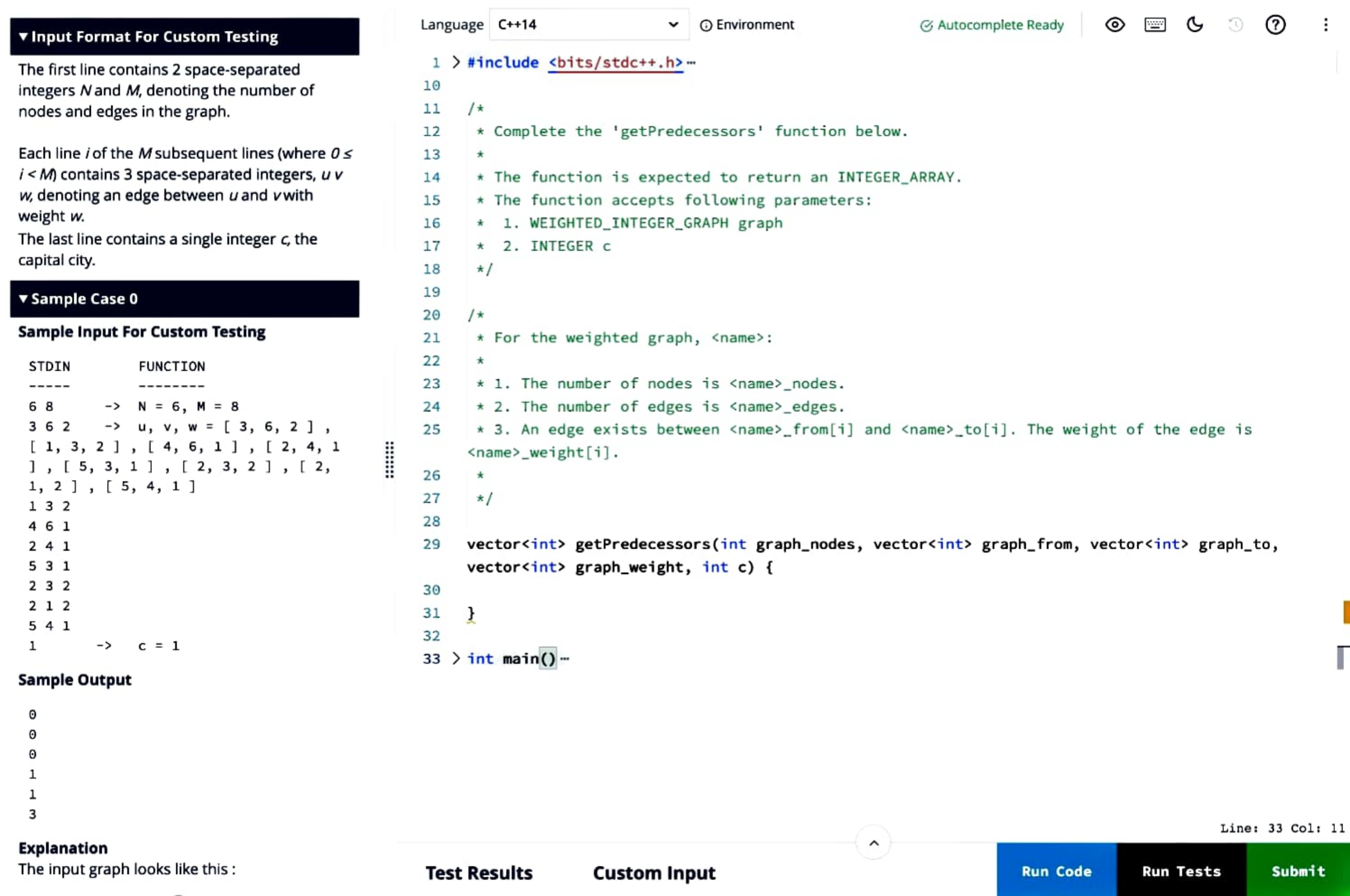
**Returns**

*int[]:* an array denoting the count of predecessors for each of the cities.

**Constraints**

- $1 \le graph\_nodes \le 200$
- $1 \le c \le graph\_nodes$
- $size(graph\_from) = size(graph\_to) = size(graph\_weight) \le [\ graph\_nodes * (graph\_nodes - 1)\ ] / 2$
- $1 \le graph\_from[i] \le graph\_nodes$
- $1 \le graph\_to[i] \le graph\_nodes$
- $1 \le graph\_weight[i] \le 10^6$

```
#include <bits/stdc++.h>···

/*
 * Complete the 'getPredecessors' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts following parameters:
 *  1. WEIGHTED_INTEGER_GRAPH graph
 *  2. INTEGER c
 */

/*
 * For the weighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
 *
 */

vector<int> getPredecessors(int graph_nodes, vector<int> graph_from, vector<int> graph_to, vector<int> graph_weight, int c) {

}

int main()···
```



### ▼ Input Format For Custom Testing

The first line contains 2 space-separated integers *N* and *M*, denoting the number of nodes and edges in the graph.

Each line *i* of the *M* subsequent lines (where *0 ≤ i < M*) contains 3 space-separated integers, *u v w*, denoting an edge between *u* and *v* with weight *w*.

The last line contains a single integer *c*, the capital city.

### ▼ Sample Case 0

**Sample Input For Custom Testing**

```
STDIN          FUNCTION
-----          --------
6 8       ->   N = 6, M = 8
3 6 2     ->   u, v, w = [ 3, 6, 2 ] ,
[ 1, 3, 2 ] , [ 4, 6, 1 ] , [ 2, 4, 1
] , [ 5, 3, 1 ] , [ 2, 3, 2 ] , [ 2,
1, 2 ] , [ 5, 4, 1 ]
1 3 2
4 6 1
2 4 1
5 3 1
2 3 2
2 1 2
5 4 1
1        ->    c = 1
```

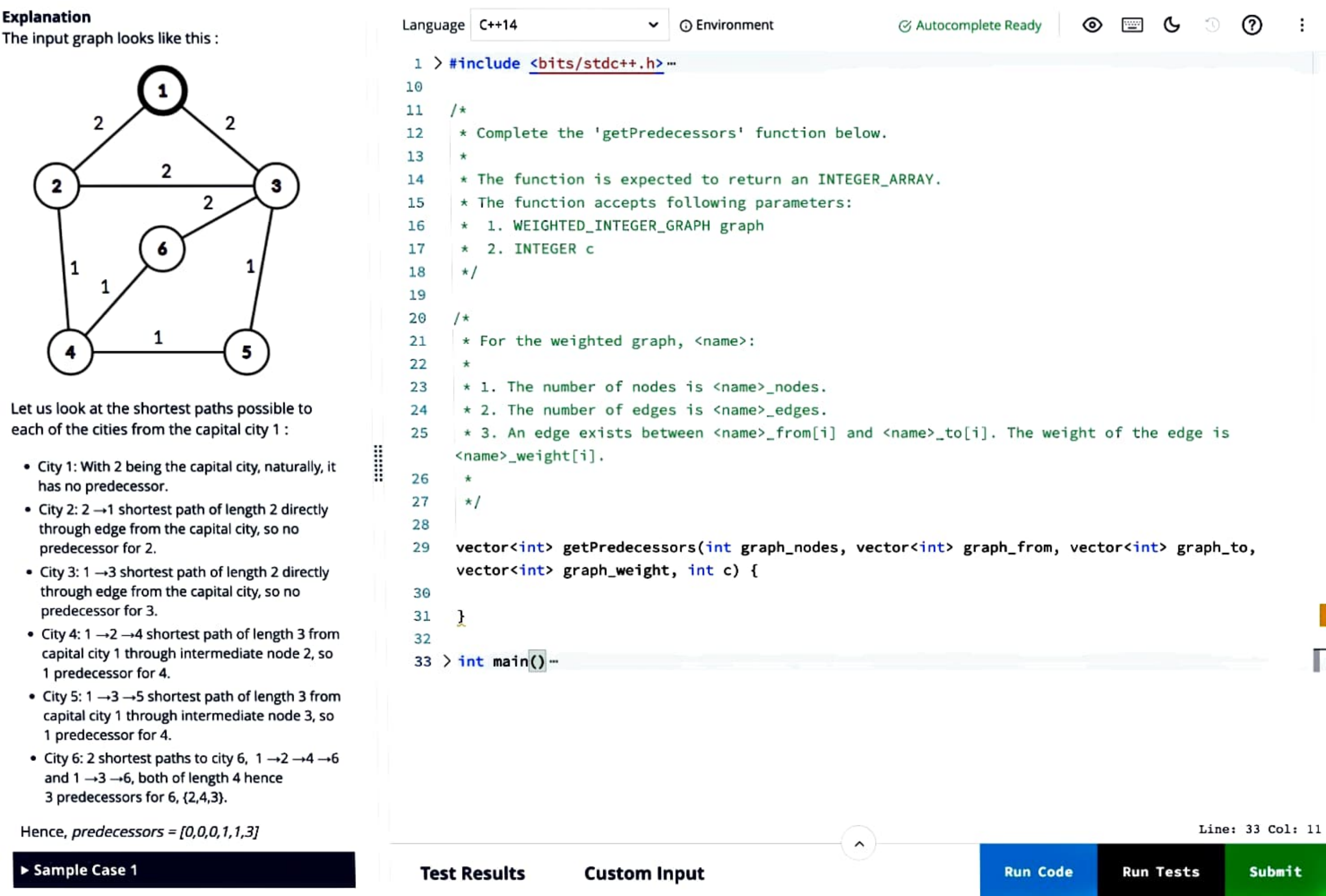
**Sample Output**

```
0
0
0
1
1
3
```

**Explanation**

The input graph looks like this :

```
#include <bits/stdc++.h>...

/*
 * Complete the 'getPredecessors' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts following parameters:
 *  1. WEIGHTED_INTEGER_GRAPH graph
 *  2. INTEGER c
 */

/*
 * For the weighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
 *
 */

vector<int> getPredecessors(int graph_nodes, vector<int> graph_from, vector<int> graph_to, vector<int> graph_weight, int c) {

}

int main()...
```



**Explanation**

The input graph looks like this :

Let us look at the shortest paths possible to each of the cities from the capital city 1 :

- City 1: With 2 being the capital city, naturally, it has no predecessor.
- City 2: 2 →1 shortest path of length 2 directly through edge from the capital city, so no predecessor for 2.
- City 3: 1 →3 shortest path of length 2 directly through edge from the capital city, so no predecessor for 3.
- City 4: 1 →2 →4 shortest path of length 3 from capital city 1 through intermediate node 2, so 1 predecessor for 4.
- City 5: 1 →3 →5 shortest path of length 3 from capital city 1 through intermediate node 3, so 1 predecessor for 4.
- City 6: 2 shortest paths to city 6, 1 →2 →4 →6 and 1 →3 →6, both of length 4 hence 3 predecessors for 6, {2,4,3}.

Hence, *predecessors = [0,0,0,1,1,3]*

```
> #include <bits/stdc++.h>⋯

/*
 * Complete the 'getPredecessors' function below.
 *
 * The function is expected to return an INTEGER_ARRAY.
 * The function accepts following parameters:
 *  1. WEIGHTED_INTEGER_GRAPH graph
 *  2. INTEGER c
 */

/*
 * For the weighted graph, <name>:
 *
 * 1. The number of nodes is <name>_nodes.
 * 2. The number of edges is <name>_edges.
 * 3. An edge exists between <name>_from[i] and <name>_to[i]. The weight of the edge is <name>_weight[i].
 *
 */

vector<int> getPredecessors(int graph_nodes, vector<int> graph_from, vector<int> graph_to, vector<int> graph_weight, int c) {

}

> int main()⋯
```