Never Return

Bimsa, a young lioness, has been banished from her home, the land of Honor Stone, by her evil uncle. He warned her to "Run away, Bimsa. Run away, and never return." Now, Bimsa must navigate her way out of Honor Stone, using her knowledge of the landmarks and trails in the area.

Your Task: You are to help Bimsa find the quickest way out of Honor Stone. She's starting from a place known as the gorge and wants to leave the area without going back on her path. Bimsa knows the location of each landmark (***N*** in total) (given as x and y coordinates) and can tell if it's inside or outside Honor Stone. She also knows the lengths of the trails between landmarks and that each landmark can have up to five connecting trails. Trails can be traveled in both directions, but Bimsa will only move from one landmark to another if it takes her farther from the gorge, following the "never return" rule.

Technical Details:

- **"Never Return" Rule:** Bimsa moves from landmark A to landmark B only if landmark B is farther from the gorge than landmark A (according to the Euclidean Distance).
- The Euclidean distance between two landmarks (x1, y1) and (x2, y2) is calculated as $\sqrt{(x1-x2)^2+(y1-y2)^2}$.

Objective: Design an efficient algorithm to find the distance of the shortest path for Bimsa to leave Honor Stone by traversing from the gorge to any landmark outside, without ever returning. The path should strictly follow the "never return" rule, moving from closer landmarks to farther ones until Bimsa is outside Honor Stone.

This task combines knowledge of geography (landmarks and distances) with algorithmic strategy to ensure Bimsa's successful escape following the constraints and rules provided.

Notes:

- gorge is always vertex 0
- Edges is **bidirectional**

Complexity

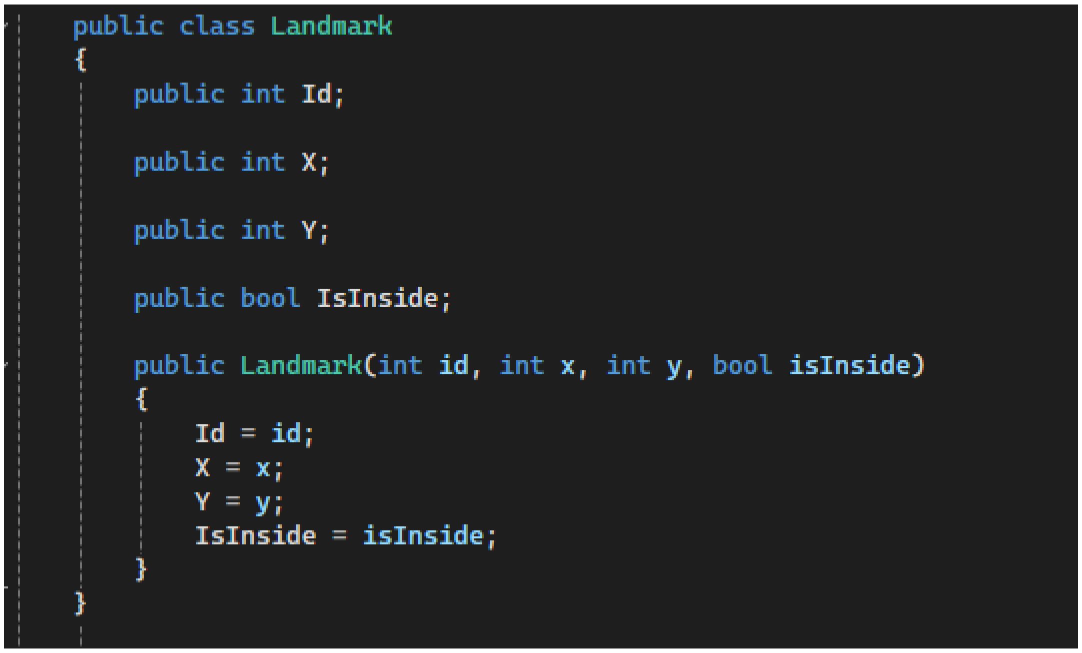
The complexity of your algorithm should be **O(N).** where N is the number of landmarks

Function to Implement

```
public static int RequiredFunction(List<Landmark> landmarks,
            List<Tuple<int, int, int>> trails, int N)
```

`PROBLEM_CLASS.cs includes this method.`

- `landmarks`: is list of `Landmark` class as show in fig 1



```
public class Landmark
{
    public int Id;

    public int X;

    public int Y;

    public bool IsInside;

    public Landmark(int id, int x, int y, bool isInside)
    {
        Id = id;
        X = x;
        Y = y;
        IsInside = isInside;
    }
}
```

Figure 1 Landmark Structure

- `trails`: List of edges in the graph, each edge is a connects two landmarks (`Tuple` where `Item1`: index of landmark1, `Item2`: index of landmark2, `Item3`: weight of this edge)
- N: the count of vertices.

<returns> the distance of the shortest path from gorge to the first node outside the honor stone.

Example

```
        n = 5;
        // Initialize landmarks
        List<GraphUtils.Landmark> landmarks1 = new List<GraphUtils.Landmark>
{
        new GraphUtils.Landmark(0, -54, 4, true),
        new GraphUtils.Landmark(1, 1885, -2334, true),
        new GraphUtils.Landmark(2, -524, 3047, true),
        new GraphUtils.Landmark(3, -1179, 6405, false),
        new GraphUtils.Landmark(4, -31, 4127, false)
};

        // Initialize edges
        List<Tuple<int, int, int>> edges1 = new List<Tuple<int, int, int>>
{
        Tuple.Create(0, 1, 3245), // edge from 0 to 1 with weight = 3245.
        Tuple.Create(1, 2, 5614),
        Tuple.Create(0, 2, 2367),
        Tuple.Create(2, 4, 9259),
        Tuple.Create(2, 3, 8561),
        Tuple.Create(1, 3, 1792),
        Tuple.Create(3, 4, 4241)
};

        Result = 5037
```

- Because the path is from 0 to 1 and from 1 to 3 = 3245 + 1792 = 5037.

C# Help

Stacks

Creation

To create a stack of a certain type (e.g. string)

```
Stack<string> myS = new Stack<string>() //default initial size

Stack<string> myS = new Stack<string>(initSize) //given initial size
```

Manipulation

1. `myS.Count` ➔ get actual number of items in the stack
2. `myS.Push("myString1")` ➔ Add new element to the top of the stack
3. `myS.Pop()` ➔ return the top element of the stack (LIFO)

Queues

Creation

To create a queue of a certain type (e.g. string)

```
Queue<string> myQ = new Queue<string>() //default initial size
```

```
Queue<string> myQ = new Queue<string>(initSize) //given initial size
```

Manipulation

1. `myQ.Count` ➔ get actual number of items in the queue
2. `myQ.Enqueue("myString1")` ➔ Add new element to the queue
3. `myQ.Dequeue()` ➔ return the top element of the queue (FIFO)

Lists

Creation

To create a list of a certain type (e.g. string)

```
List<string> myList1 = new List<string>() //default initial size

List<string> myList2 = new List<string>(initSize) //given initial size
```

Manipulation

1. `myList1.Count` ➔ get actual number of items in the list
2. `myList1.Sort()` ➔ Sort the elements in the list (ascending)
3. `myList1[index]` ➔ Get/Set the elements at the specified index
4. `myList1.Add("myString1")` ➔ Add new element to the list
5. `myList1.Remove("myStr1")` ➔ Remove the 1st occurrence of this element from list
6. `myList1.RemoveAt(index)` ➔ Remove the element at the given index from the list
7. `myList1.Contains("myStr1")` ➔ Check if the element exists in the list

Dictionary (Hash)

Creation

To create a dictionary of a certain key (e.g. string) and value (e.g. array of strings)

```
//default initial size
Dictionary<string, string[]> myDict1 = new  Dictionary<string, string[]>();

//given initial size
Dictionary<string, string[]> myDict2 = new  Dictionary<string, string[]>(size);
```

Manipulation

1. `myDict1.Count` ➔ Get actual number of items in the dictionary
2. `myDict1[key]` ➔ Get/Set the value associated with the given key in the dictionary
3. `myDict1.Add(key, value)` ➔ Add the specified key and value to the dictionary
4. `myDict1.Remove(key)` ➔ Remove the value with the specified key from the dictionary
5. `myDict1.ContainsKey(key)` ➔ Check if the specified key exists in the dictionary

Creating 1D array

```
int [] array = new int [size]
```

Creating 2D array

```
int [,] array = new int [size1, size2]
```

Length of 1D array

```
int arrayLength = my1DArray.Length
```

Length of 2D array

```
int array1stDim = my2DArray.GetLength(0)

int array2ndDim = my2DArray.GetLength(1)
```

Sorting single array

Sort the given array in ascending order

```
Array.Sort(items);
```

Sorting parallel arrays

Sort the first array "master" and re-order the 2nd array "slave" according to this sorting

```
Array.Sort(master, slave);
```