Skating Travel

Description

Ali Baba decides to go on a skating travel in the alpine mountain. He has stolen a pair of skis and a trail map listing the mountain's surfaces and slopes (*N* in total), and he wants to ski from surface *S* to surface *T* where a treasure is exists.

- Each **surface** $s_i \in S$ has an integer **elevation** e_i above sea level.
- Each **slope** (s_i, s_j, l_{ij}) connects a pair of surfaces s_i and s_j with a **monotonic** trail (strictly decreasing or increasing in elevation) with positive integer length l_{ij}. Each slope is considered a **bidirectional** trail.

Ali Baba doesn't have time to ski uphill, so he will only traverse slopes so as to decrease his elevation.

Given Ali Baba's map, describe an O(N)-time algorithm to find the minimum distance he must ski to reach the treasure.

Complexity

The complexity of your algorithm should be **O(N)**.

Function to Implement

```
public static int RequiredFunction(Dictionary<string, int> vertices,
Dictionary<KeyValuePair<string, string>, int> edges, string startVertex)
```

`PROBLEM_CLASS.cs includes this method.`

- "vertices": container of surfaces in the graph (where **`key: vertexName, value: elevation value`**)
- "edges": container of trails in the graph (where **`key: <surface1,surface2>, value: trail length`**)
- "startVertex": name of the start vertex to begin from it which is always denoted as "S".

<returns> the minimum valid distance from source "S" to target "T".

Example

1-

```
Dictionary<string, int> vertices1 = new Dictionary<string, int>();
vertices1["S"] = 10;
vertices1["A1"] = 8;
vertices1["A2"] = 9;
vertices1["A3"] = 4;
vertices1["A4"] = 12;
vertices1["T"] = 2;

connection11 = new KeyValuePair<string, string>("S", "A2");
connection12 = new KeyValuePair<string, string>("S", "A1");
connection13 = new KeyValuePair<string, string>("S", "A4");
connection14 = new KeyValuePair<string, string>("A3", "A1");
connection15 = new KeyValuePair<string, string>("A3", "T");
connection16 = new KeyValuePair<string, string>("A2", "T");
connection17 = new KeyValuePair<string, string>("A4", "T");

edges1[connection11] = 9;
edges1[connection12] = 5;
edges1[connection13] = 2;
edges1[connection14] = 3;
edges1[connection15] = 1;
edges1[connection16] = 4;
edges1[connection17] = 3;

expected1 = 9;
```

2-

```
Dictionary<string, int> vertices2 = new Dictionary<string, int>();
vertices2["S"] = 12;
vertices2["A1"] = 8;
vertices2["A2"] = 2;
vertices2["A3"] = 9;
vertices2["T"] = 4;

connection21 = new KeyValuePair<string, string>("S", "A1");
connection22 = new KeyValuePair<string, string>("A2", "A1");
connection23 = new KeyValuePair<string, string>("A2", "T");
connection24 = new KeyValuePair<string, string>("S", "T");
connection25 = new KeyValuePair<string, string>("A3", "S");
connection26 = new KeyValuePair<string, string>("A3", "T");
```

```
edges2[connection21] = 1;
edges2[connection22] = 1;
edges2[connection23] = 1;
edges2[connection24] = 12;
edges2[connection25] = 5;
edges2[connection26] = 6;

expected2 = 11;
```

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);
```