Programming Problem: Prim's Algorithm with Tunable Heap

[15 points for the programs, 5 points for the analysis.]

For this problem, you get to implement Prim's algorithm using the heap you created for homework 2. By adjusting the branching factor based on the density of the graph, we'll be able to achieve a slightly better asymptotic runtime than Prim's algorithm with a binary heap.

Implementation Language

As with previous assignments, you get to implement your solution in Java, C or C++, whichever language you prefer. The name of your source file will depend on your language:

- For Java : prim.java
- For C : prim.c
- For C++ : prim.cpp

When we test your solutions, we'll compile and run them on an EOS Linux machine using the following commands. Be sure to try these out yourself to make sure you program will compile and work when we try it out. Your program will read its input from standard input (i.e., from the terminal), but, like in the following examples, we'll use I/O redirection to get it to read from a file instead.

- For Java

```
javac prim.java
java prim < input-3.txt
```

- For C

```
gcc -Wall -std=c99 -g -O2 -o prim prim.c
./prim < input-3.txt
```

- For C++

```
g++ -Wall -g -O2 -o prim prim.cpp
./prim < input-3.txt
```

Input Format

The input will describe a weighted, undirected graph. The first line will contain two values. The first is a positive integer, V, giving the number of vertices in the graph. The second is a non-negative integer, E giving the number of edges.

The next E lines of input each describe one of the edges. An edge is given as three integers a b and w, where a and b are the indices of two vertices connected by the edge. Vertices are indexed from 0 to $V - 1$. The w value is a postiive integer giving the weight of the edge.

The following shows sample input, `input-1.txt`. This file describes a graph with 6 vertices and 9 edges:

```
6 9
0 2 4
0 1 8
2 3 1
3 1 3
3 4 2
```

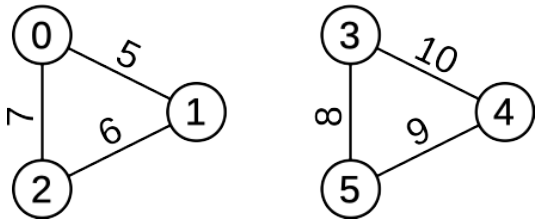
```
1 4 10
2 5 9
3 5 6
5 4 7
```

The number of vertices, the number of edges and the total weight of a minimum-weight spanning tree will all fit in a signed, 32-bit integer. There will be at most one edge between any pair of vertices, and no edge will connect a vertex with itself.

As shown by sample input, `input-2.txt`, the graph may not be connected. The input file for this graph is as follows:

```
6 6
0 1 5
1 2 6
2 0 7
3 4 10
4 5 9
5 3 8
```

This describes a graph that looks like the following.



Forest of Minimum-Weight Trees

Since the graph may have multiple disconnected components, you'll really be finding a forest of trees, consisting of a minimum-weight tree for each component.

To find such a forest, you'll keep a flag for each vertex, indicating whether it is part of a spanning tree. Initially, all these flags will be false.

You'll iterate over each vertex in the graph. If the vertex is already part of a spanning tree, you can move on to the next vertex. If it's not part of a spanning tree, you will use it as a root vertex to grow another spanning tree. So, each time you start growing a spanning tree, you'll typically find a group of vertices that are reachable from the root (or, maybe just the root itself), but you may not reach every vertex in the graph. After building one tree, you'll move on to the next vertex to see if it's part of a different component, and, if it is, grow another tree from there. When you're done, you should have a tree for every connected component. Taken together, we'll call all these trees a forest of minimum-weight spanning trees.

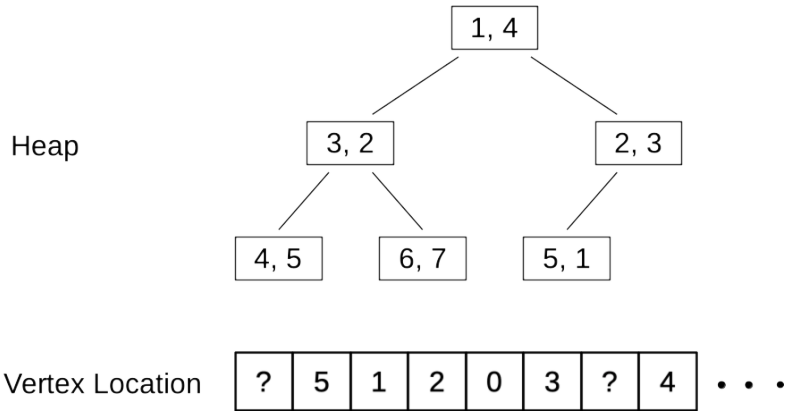
Heap Implementation

When we wrote the heap for homework 2, we were thinking ahead to how we might use it for other algorithms. Recall that elements of your heap have two parts, a key that's used to maintain the heap order and a value that can represent some integer associated with that key. For this application, were going to use that value to keep a vertex. Whenever we insert a vertex in the heap, its key value will indicate the best known cost for adding some vertex, i, to the spanning three, and the value will be the index, i, of the vertex.

We'll need to make one important modification to our heap for this program. In Prim's algorithm,

whenever we find a lower-cost edge via which some vertex could be added to the spanning tree, we need to do a decrease key for that vertex. You'll add a new decreaseKey() function to support this, but we'll also need an efficient way to find the location of a vertex in the heap. This will let us go directly to the heap element for that vertex, decrease its key and move it up in the heap as needed.

For this, we'll need one more array (or vector, or ArrayList) of V integers. Element i in this array will maintain the current location of vertex i in the heap. This is illustrated in the following figure. Each element of the heap shows a key, followed by the vertex with that key. The array below the heap shows where each vertex is in the heap. For example, vertex 1 is at element 5 in the heap and vertex 7 is at element 4 (recall, we counted heap locations from zero for this problem). Some vertices may not currently be in the heap at all. I just marked these with a question mark in the figure.



Since elements of the heap can move around in response to heap insertions, remove-min operations and decrease-key operations, we'll need to keep the list of vertex locations updated as the heap is modified. For example, if we do a decrease-key on vertex 7, reducing its key to 2 instead of 7, we'll need to update the location of this vertex, since it's now element 1 of the heap. We'll also need to update the location of vertex 2 since it's now in position 4 in the heap.

One part of your heap can get simpler. For this application, your heap no longer needs to keep a count of the key comparisons. You can leave that code in (and just ignore the count) or remove it if you want.

Heap Implementation Head Start

If you had trouble implementing the heap for homework 2, we don't want this to prevent success on this homework. We'll provide working sample implementations of the heap in C, C++ and Java. You'll have to understand and modify these implementations if you want to use them, but this might get you started more quickly if you're not happy with your solution from the earlier assignment.

We'll post a note on Piazza when these sample implementations are available, with instructions for how you can download them.

Heap Branching Factor

We're going to choose a heap branching factor that's depends on the density of the graph. Specifically, we'll use a branching factor of b where b is the smallest power of 2 that's at least E/V (and no smaller than 2, since that's the smallest branching factor our heaps were expected to support).

Program Output

As output, just print one line to standard output, giving three values. First, print the branching factor your program used for its heap. Then, after a space, report the the number of trees your program finds in its forest of minimum-weight trees. Finally, print the total weight of all the minimum-weight spanning trees in the forest.

Sample Execution

We're providing five sample inputs with this problem. There are some smaller ones to help you check the behavior of your program and some larger ones to help you see a little bit of the performance costs. If you run your program on these inputs, here's what you should expect:

- For Java

```
java prim < input-1.txt
2 1 16
java prim < input-2.txt
2 2 28
java prim < input-3.txt
4 1 39
java prim < input-4.txt
4 8 853
java prim < input-5.txt
32 11 101670
```

- For C or C++

```
./prim < input-1.txt
2 1 16
./prim < input-2.txt
2 2 28
./prim < input-3.txt
4 1 39
./prim < input-4.txt
4 8 853
./prim < input-5.txt
32 11 101670
```

Algorithm Analysis

Write up a short, worst-case analysis of the performance of your Prim's algorithm implementation. In your analysis, use V to represent the number of vertices and let E represent the number of vertices. You can assume the branching factor for your heap is exactly E/V (although, for our implementation, we chose to round up to the nearest power of two).

Submitting Your Work

In Moodle, you'll see an assignment named `Assignment 4`. Submit the source code to your program using this link. For your analysis, submit that to a GradeScope assignment with the name `Program 4 Analysis`.