CS 330 – Minimum Spanning Trees

deadline: Monday 10/21/2019 at 11:59 pm

In this exercise you will implement an algorithm for finding the minimum spanning tree in an undirected graph with weighted edges. The problem is in Section 4.5 of *Algorithm Design.* There are several algorithms for finding these trees: two described with implementation details in the book are Kruskal's Algorithm and Prim's Algorithm. (The implementation of Prim's is similar to that of Dijkstra's Algorithm.) Whatever you choose, your algorithm must run in $O(m \log n)$ time for m edges and n vertices. We've set rough upper bounds on execution time; if your code exceeds these, you'll see the limit and your code's running time.

Submission Format

On Gradescope submit a file (either `minimum_spanning_tree.py` or `minimum_spanning_tree.java`, names must match exactly) that reads a text file `input` and creates a text file `output` containing your solution. **Important:** you may submit other files containing additional functions and classes. The autograder will put your code in the same directory as `input` and call it from the command line. Your code must create and write `output` in that directory.

Since the focus of this exercise is on the algorithm, you may import a built-in priority queue, such as Python's `heapq` or Java's `PriorityQueue`. Note that these implementations may not have all the methods the textbooks assumes. You are also free to implement your own priority queue. The textbook discusses using heaps as priority queues on page 64.

Input

The file `input` contains $m + 2$ lines specifying a graph with weighted edges. The first two lines give the number of vertices and the number of edges, respectively. The vertices are indexed $0, 1, \ldots, n - 1$. Every row after the first two will specify an edge by giving the starting node, the ending node, and the weight. The weights will all be integers. You may assume the edge weights are distinct and the graph is connected. The edges will be given only once: for an edge from u to v with weight w, you will see either `u,v,w` or `v,u,w`, but not both.

The lines of `input` will contain the following information:

1. n, the number of vertices.
2. m, the number of edges.
3. The starting node of edge 0, a comma, the ending node of edge 0, a comma, the weight on edge 0.
4. The starting node of edge 1, a comma, the ending node of edge 1, a comma, the weight on edge 1.
5. $\ldots$
6. The starting node of edge $m - 1$, a comma, the ending node of edge $m - 1$, a comma, the weight on edge $m - 1$.

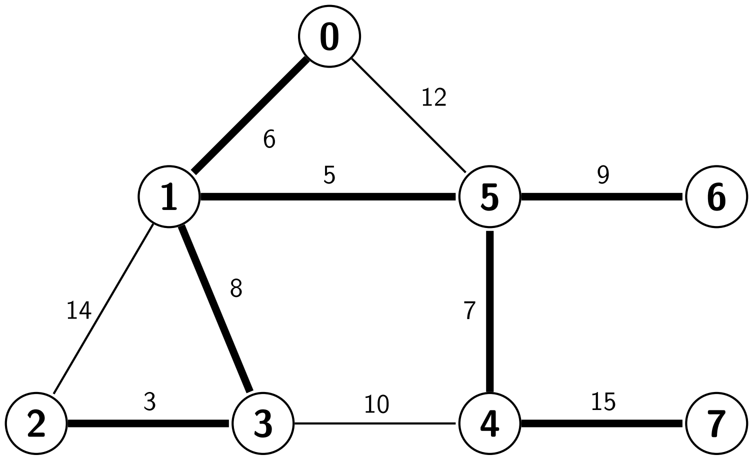
We have uploaded files corresponding to the visible test cases on Gradescope. They may help you test and debug locally. Note: passing these test cases does not guarantee your code is correct!

Output

Your algorithm must create a text file `output` specifying the adjacency list of the minimum spanning tree. The file must contain n lines giving the neighbors of each vertex. The top line will list the neighbors of vertex 0, the next line the neighbors of vertex 1, and so on. Since you return a spanning tree, all of the vertices will have at least one neighbor. The graph is undirected, so remember to put symmetric edges in the graph: if (u, v) is an edge, so is (v, u). You don't need to include weight information in the output file.

Example

Consider the following directed graph, taken from the lecture slides. The edges in the minimum spanning tree are in bold.



This graph corresponds to the following input file. Note that each edge only appears once in the list. The edges may appear in any order.

```
input:

8
10
0,1,6
0,5,12
5,1,5
1,2,14
1,3,8
2,3,3
3,4,10
7,4,15
4,5,7
5,6,9
```

Note that neither `input` nor `output` have blank lines at the top; that's just how we're presenting it. The output gives the adjacency list for the minimum spanning tree:

```
output:

1
0,5,3
3
1,2
5,7
1,6,4
5
4
```