Problem 2: Minimum Spanning Tree (mst)

The C++ standard template library provides us with a large number of containers and data structures that we can go ahead and use in our programs. We'll be learning how to use a priority queue, which is available in the <queue> header.

The problem we can solve using a priority queue is that of computing a *minimum spanning tree*. Given a fully connected undirected graph where each edge has a weight, we would like to find the set of edges with the least total sum of weights. This may sound abstract; so here's a concrete scenario:

You're a civil engineer and you're trying to figure out the best way to arrange for internet access in your small island nation of C-Land. There are N $(3 \leq N \leq 100,000)$ towns on your island connected by various roads and you can walk between any two towns on the island by traversing some sequence of roads.

However, you've got a limited budget and have determined that the cheapest way to arrange for internet access is to build some fiber-optic cables along existing roadways. You have a list of the costs of laying fiber-optic cable down along any particular road, and want to figure out how much money you'll need to successfully complete the project–meaning that, at the end, every town will be connected along some sequence of fiber-optic cables.

Luckily, you're also C-Land's resident computer scientist, and you remember hearing about Prim's algorithm in one of your old programming classes. This algorithm is exactly the solution to your problem, but it requires a priority queue...and ta-da! Here's the C++ standard template library to the rescue.

If this scenario is still not totally clear, look at the sample input description on the next page. Our input data describing the graph will be arranged as an adjacency list: for every node in the graph (town in the country), we'll have a list of the nodes (towns) it's connected to and the weight (cost of building fiber-optic cable along).

$$\begin{aligned} &\texttt{adj[0]} \rightarrow (1, 1.0)\ (3, 3.0) \\ &\texttt{adj[1]} \rightarrow (0, 1.0)\ (2, 6.0)\ (3, 5.0)\ (4, 1.0) \\ &\qquad\vdots \end{aligned}$$

This data structure might be a pain to allocate and keep track of everything in C. The C++ STL will simplify things. First, our adjacency list can be represented as a list of C++ vectors, one for each node. To further simplify and abstract things, we've created a wrapper class called `AdjacencyList` that you can use; it already has a method written to help with reading the input.

Input Format

Line 1: A single integer N indicating the number of vertices in the graph.
Lines $2 \ldots N$: Line $i+2$ contains an integer m_i, the number of nodes adjacent to node with ID i, followed by m_i space-separated pairs of integers v_{ij} and doubles w_{ij} – representing the IDs of adjacent nodes and the weight of the edge between the two nodes respectively.

Sample input (file mst.in)

```
6
2 1 1.0 3 3.0
4 0 1.0 2 6.0 3 5.0 4 1.0
3 1 6.0 4 4.0 5 2.0
3 0 3.0 1 5.0 4 1.0
4 1 1.0 2 4.0 3 1.0 5 4.0
2 2 2.0 4 4.0
```

Input Explanation

We can visualize this graph as in Figure 1; let $A, B, C, \ldots$ represent the nodes with IDs $0, 1, 2, \ldots$ respectively. Looking at our input file, the we can see that the second line `2 1 1.0 3 3.0` describes the nodes adjacent to vertex 0 (or A in the diagram): node 0 is connected by an edge of length 1.0 to node 1 and an edge of length 3.0 to node 3. On the right, we can see a minimum spanning tree for

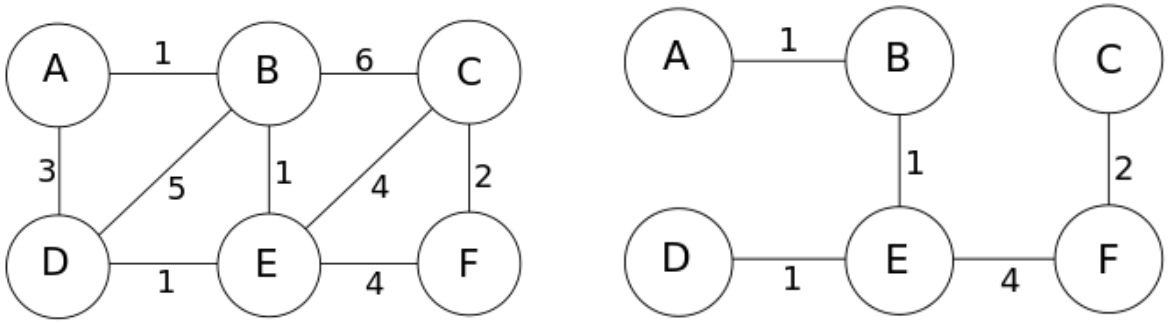

Figure 1: The full graph on the left and the minimum spanning tree on the right.

our graph. Every vertex lies in one totally connected component, and the edges here sum to $1.0 + 1.0 + 1.0 + 4.0 + 2.0 = 9.0$, which will be our program's output.

Output Format

Line 1: A single floating-point number printed to at least 8 decimal digits of precision, representing the total weight of a minimum spanning tree for the provided graph.

Sample output (file mst.out)

```
9.00000000
```

Template Code

If you take a look in the provided template file at http://web.mit.edu/6.s096/www/hw/2/mst.zip, you'll see some data structures already written for you.

```
#include <vector>

class State {
  size_t _node;
  double _dist;
public:
  State( size_t aNode, double aDist ) : _node{aNode}, _dist{aDist} {}
  inline size_t node() const { return _node; }
  inline double dist() const { return _dist; }
};

class AdjacencyList {
  std::vector< std::vector<State> > _vert;
  AdjacencyList() = delete;
public:
  AdjacencyList( std::istream &input );
  inline size_t size() const { return _vert.size(); }
  inline const std::vector<State>& vert( size_t node ) const {
    return _vert[node];
  }
  void print();
};
```

Some questions to ask yourself for understanding:

- What does `AdjacencyList() = delete;` mean? Why did we do that?
- This is a fairly complicated line:
  `inline const std::vector<State>& vert( size_t node ) const.`
  Justify or question each use of `const`.
- Why don't we need to write our own destructor for the `AdjacencyList` class?
- How large is a single instance of the `State` class in memory, most likely?

Think these questions through and ask about anything that you're unsure of on Piazza.