The C++ standard template library provides many containers and data structures. In this project we will use a priority queue, which is available in the `<queue>` header.

We want to use a priority queue to compute a minimum spanning tree. That is, given a fully connected undirected weighted-edge graph, we want to find the set of edges with the least total sum of weights which provides access to all the vertices.

To make this more concrete, consider this scenario:
You're a pepper farmer and you're trying to figure out the best way to deliver peppers to towns in Pepper-Land. There are $N(3 \leq N \leq 250,000)$ towns in Pepper-Land connected by $N \leq M \leq 250,000$ various roads and you can walk between any two towns in Pepper-Land by traversing some sequence of roads.

However, your budget is limited budget and you're seeking the cheapest way to arrange for pepper delivery, which you've decided is to build some pneumatic pepper shoots along existing roadways. You have a list of the costs of building pepper shoots along any particular road, and want to figure out how much it'll cost to complete the project-meaning that, at the end, every town will be connected along some sequence of pneumatic pepper shoots.

Luckily, you're also Pepper-Lands 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.

Look at the sample input that follows. The input data describing the graph will be arranged as a list of edges (roads and their pepper shoot costs) which we'll covert to 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 pepper shoots along).

```
adj[0] -> (1, 1.0) (3, 3.0)
adj[1] -> (0, 1.0) (2, 6.0) (3, 5.0) (4, 1.0)
.
.
.
```

The C++ STL simplifies things. First, the adjacency list can be represented as a list of `vectors`, one for each node. To further simplify and abstract things, we'll make a wrapper class called `AdjacencyList` that you can use; it already has a method written to help with reading the input.

Resource Limits

Your program will be allowed up to 3 seconds of runtime and 32MB of RAM.

Input Format

Line 1: Two space-separated integers: N, the number of nodes in the graph, and M, the number of edges.

Lines $2 \ldots M$: Line i contains three space- separated numbers describing an edge: s_i and t_i , the IDs of the two nodes involved, and w_i, the weight of the edge.

Sample input (file mst.in)

```
6 9
0 1 1.0
1 3 5.0
3 0 3.0
3 4 1.0
1 4 1.0
1 2 6.0
5 2 2.0
2 4 4.0
5 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 011.0 describes edge between A and B of weight 1.0 in our diagram, the third line 135.0 describes the edge between B and D of weight 5.0, and so on. On the right, we can see a minimum spanning tree for our graph. Every

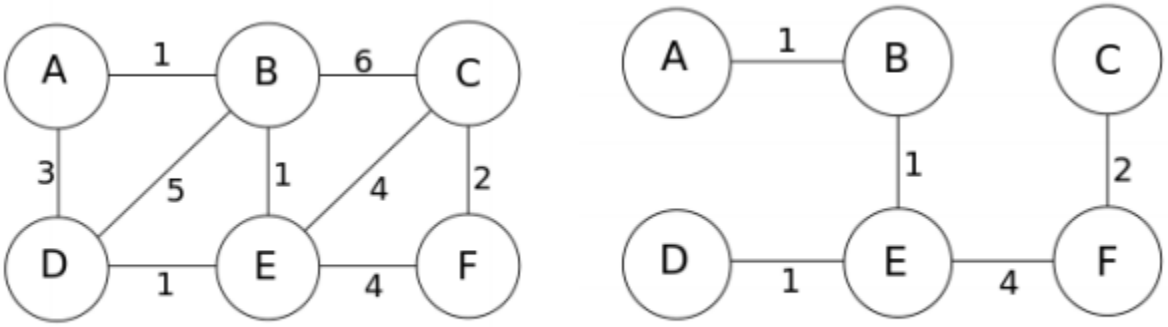

Figure 1: The full graph on the left and the minimum spanning tree on the right.

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 the 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 provided in the file mst.data.zip as a basis for your program, 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 dont 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 in class.