Computer Science 2530
Remarks on Assignment 5

Approach assignment 5 by trying to find the simplicity in it. Work by successive refinement. Build a little, then test that before moving on.

Storing a graph

You will need a structure type for an edge. It holds two vertex numbers and a weight. All three numbers are integers.

Add a parameterless constructor for type **Edge** that sets all parts to hold 0, and possibly another constructor that takes two vertex numbers and a weight and installs them.

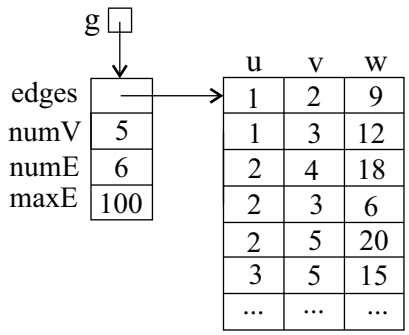
You will also need a structure type for a graph. It should hold:

1. the number of vertices;
2. the number of edges;
3. a pointer to an array of edge structures;
4. the physical size of the array of edges.

The assignment shows a graph described by the following input.

```
5
1 2  9
1 3 12
2 4 18
2 3  8
2 5 20
3 5 15
0
```

Here is a diagram of the representation of that graph.



Notice that the first edge is g->edges[0], and the 'u' component of the first edge is g->edges[0].u.

Add a constructor for type **Graph** that takes the number of vertices and the maximum number edges to allow, *and that is all*. The constructor should allocate the array of edges, and it should set the current number of edges to 0.

Be careful not to confuse the physical size of the edge array with its logical size, which is the number of edges that it currently holds.

Reading and writing a graph

The input starts with the number of *vertices*. The number of edges is not explicitly in the input. **Do not change the input format to suit yourself. If you change the input format, then you are not solving the assignment that you have been assigned.**

Write **insertEdge** so to add an edge to the graph. Notice that the next edge goes at index g->numE. **InsertEdge** must to the whole job of inserting an edge, and that includes adding 1 to the current number of edges.

Write **readGraph** so that it:

1. reads the number of vertices;
2. reads each edge until it reads 0, inserting the edges using **insertEdge**.

Notice that the parameter, e, of **readGraph** is the maximum number of edges to be allowed. It is not the actual number of edges in the graph.

When **main** calls **readGraph**, it should choose a maximum number of edges that is at least 100. But also make it so that you can change the maximum number of edges allowed without modifying the definition of **main**. Instead, define a named constant near the beginning of the program, where it is easy to find.

Write **writeGraph**. Write bf main so that it just reads a graph and then writes the graph.

Sorting the edges

Write sortEdges to sort the array of edges in a graph g. Start with:

```
#include <cstdlib>
typedef int (*QSORT_COMPARE_TYPE)(const void*, const void*);

...
// compareEdges(A,B) returns a result that is
//    > 0 if edge A has a larger weight than B
//    < 0 if edge A has a smaller weight than B
//    = 0 if edges A and B have the same weight.

int compareEdges(const Edge* A, const Edge* B)
{
  return A->weight - B->weight;
}
```

Read about qsort in item 7 of the refinement plan. In the call

```
qsort(E, N, sizeof(Edge), (QSORT_COMPARE_TYPE) compareEdges);
```

E must be the array of edges and N must be the current number of edges.

Write **minimalSpanningTree** so that it only sorts the array of edges and returns g. Modify **main** so that it calls **minimalSpanningTree** and writes the resulting graph (using **writeGraph**!). You should see the original graph, but with the edges sorted by weight.

Computing a minimal spanning tree

Now you are ready to flesh out **minimalSpanningTree**. Using the **Graph** constructor, build a new graph that will become the minimal spanning tree of g. A tree with n vertices always has $n - 1$ edges, so you know exactly how large the array of edges needs to be. Don't make it too large or too small. Initially, your minimal spanning tree graph has no edges.

You will need an equivalance relation manager, where a pair u and v of vertices are in the same group if there is a path from u to v. Since there are initially no edges, each vertex is in a group by itself.

Loop through the edges of g, in order from smallest weight to largest weight. When you encounter an edge between u and v in g, add it to the minimal spanning tree (using **insertEdge**!) provided u and v are not already connected to one another. Use the equivalence relation manager.

Pay attention to the difference between vertices and edges. There g->numE edges.

Finishing up

In the past, some students have inexplicably written **minimalSpanningTree**(g) so that it destroys g. They did it because they did not want to create a local variable inside **minimalSpanningTree**. Don't do that!

Temporarily modify **main** so that it writes the original graph a second time, after computing the minimal spanning tree. Is the original graph the same as it was (except that the edges have been sortted into increasing order by weight)? Once you are satisfied that you have not destroyed the original graph, remove the extra echo of the graph.