Submit via blackboard, both the write-up and code should be included. Try to make sure your code can compile and run, you will lose most of the scores if your code can not run. If there are bugs that you can't fix, try to address them in your write-up. If you prefer to do late submission and take the penalty, please inform TA and professor before the deadline.

Overview:
The purpose of this assignment is to experience some of the problems involved with implementing an algorithm (in this case, a minimum spanning tree algorithm) in practice. As an added benefit, we will explore how minimum spanning trees behave in random graphs.

Assignment:
You may work in groups of two, or by yourself. Both partners will receive the same grade and turn in a single joint report.

We will be considering *complete, undirected* graphs. A graph with n vertices is complete if all possible $\binom{n}{2}$ edges are present in the graph.

Consider the following types of graphs:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random on $[0, 1]$.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are (x, y), with x and y each a real number chosen uniformly at random from $[0, 1]$.) The weight of an edge is just the Euclidean distance between its endpoints.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit cube (3 dimensions) and hypercube (4 dimensions). As with the unit square case above, the weight of an edge is just the Euclidean distance between its endpoints.

Your first goal is to determine in each case how the expected (average) weight of the minimum spanning tree grows as a function of n. This will require implementing an MST algorithm, as well as procedures that generate the appropriate random graphs. (You should check to see what sorts of random number generators are available on your system, and determine how to seed them, say with a value from the machine's clock.) You may implement any MST algorithm (or algorithms!) you wish; however, I suggest you choose carefully.

For each type of graph, you must choose several values of n to test. For each value of n, you must run your code on several randomly chosen instances of the same size n, and compute the average value for your runs. Plot your values vs. n, and interpret your results by giving a simple function $f(n)$ that describes your plot. For example, your answer might be $f(n) = \log n$, $f(n) = 1.5\sqrt{n}$, or $f(n) = \frac{2n}{\log n}$. Try to make your answer as accurate as possible; this includes determining the constant factors as well as you can. On the other hand, please try to make sure your answer seems reasonable.

Code setup:

So that we may test your code ourselves as necessary, please make sure your code accepts the following command line form:

randmst 0 numpoints numtrials dimension

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. The value numpoints is n, the number of points; the value numtrials is the number of runs to be done; the value dimension gives the dimension. (Use dimension = 2 for the square, and 3/4 for cube and hypercube; use dimension = 0 for the case where weights are assigned randomly. Notice that dimension 1 is just not that interesting.) The output for the above command line should be the following:

average numpoints numtrials dimension

where average is the average minimum spanning tree weight over the trials.

Please pay attention to the following requirements. In order to grade appropriately, our objective is to ensure that we can run the programs without any special per-student attention. Hence we require:

- The code should run on our department's server, even if you code on another system.
- We expect an executable with the code as described above (input as described, answers should go to standard output as described).
- The code should compile with make; no instructions for humans.

What to hand in: Besides codes, your group should hand in a single well organized and clearly written report describing your results. For the first part of the assignment, this report must contain the following quantitative results (for each graph type):

- A table or graph listing the average tree size for several values of n.
- A description of your guess for the function $f(n)$.

Run your program for $n = 16; 32; 64; 128; 256; 512; 1024; 2048; 4096; 8192; 16384$ and larger values, if your program runs fast enough. Run each value of n at least five times and take the average. (Make sure you re-seed the random number generator appropriately!)

In addition, you are expected to discuss your experiments in more depth. This discussion should reflect what you have learned from this assignment; the actual issues you choose to discuss are up to you. Here are some possible suggestions for the second part:

- Which algorithm did you use, and why?
- Are the growth rates (the $f(n)$) surprising? Can you come up with an explanation for them?
- How long does it take your algorithm to run? Does this make sense? Do you notice things like the cache size of your computer having an effect?
- Did you have any interesting experiences with the random number generator? Do you trust it?

You will also be submitting your code via blackboard, so that we may test it as necessary. The submission process will be posted on line.

Your grade will be based primarily on the correctness of your program and your discussion of the experiments. Other considerations will include the size of n your program can handle. Please do a careful job of solid writing in your writeup. Length will not earn you a higher grade, but clear descriptions of what you did, why you did it, and what you learned by doing it will go far.

Hints:

To handle large n, you may want to consider simplifying the graph. For example, for the graphs in this assignment, the minimum spanning tree is extremely unlikely to use any edge of weight greater than $k(n)$, for some function $k(n)$. We can estimate $k(n)$ using small values of n, and then try to throw away edges of weight larger than $k(n)$ as we increase the input size. Notice that throwing away too many edges may cause problems. Why will throwing away edges in this manner never lead to a situation where the program returns the wrong tree?

You may invent any other techniques you like, as long as they give the same results as a non-optimized program. Be sure to explain any techniques you use as part of your discussion and attempt to justify why they should give the same results as a non-optimized program!