This is a test. You should do this test entirely on your own, without discussing it or working on it with anyone. You can use your textbook and notes, but you should not use any other books or resources, including the Internet. There will be time in class on Monday and Wednesday to ask for hints or clarification, when everyone can hear the answers. If I answer any substantive questions during my office hours or by email, I will share my answers with the class. The test is due in class next Friday, April 5.

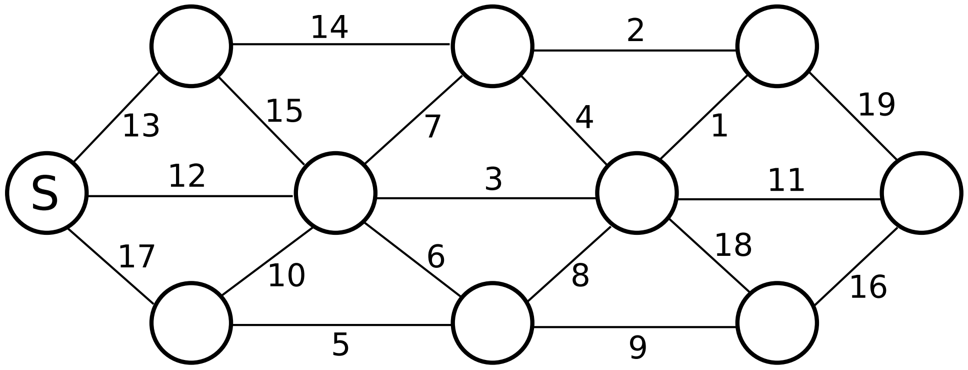
For question 1, you should mark your answers on this sheet. You should attach your answers to questions 2 through 5. (Word-processed responses would be appreciated, but handwritten are acceptable.)

Each of the five questions on this test counts for twenty points.

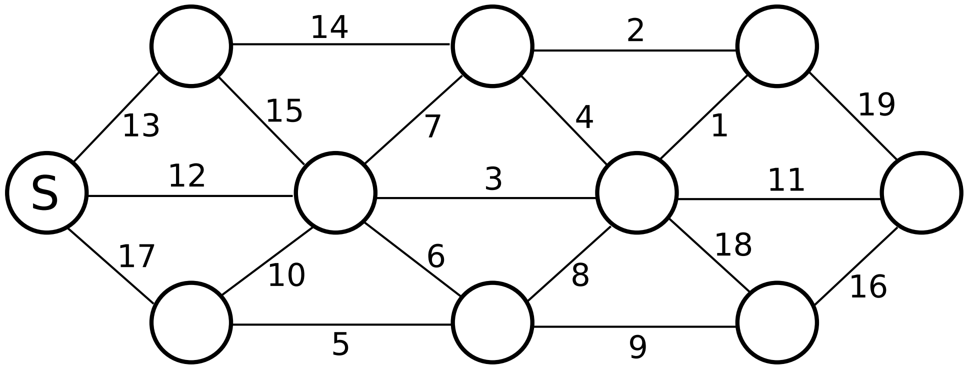
Sign the test on the following line. If your signature is not readable, please also print your name. By signing, you affirm that the work that you are submitting is entirely your own work, as discussed above.

1. This problem asks you to apply three graph algorithms that we have studied. It is meant to be fairly easy.

a) Suppose that **Dijkstra's algorithm** is applied to find shortest paths from the vertex on the left (marked with an "S") to all the other vertices in the graph. Mark the **first six** edges that are added to the graph by Dijkstra's algorithm.

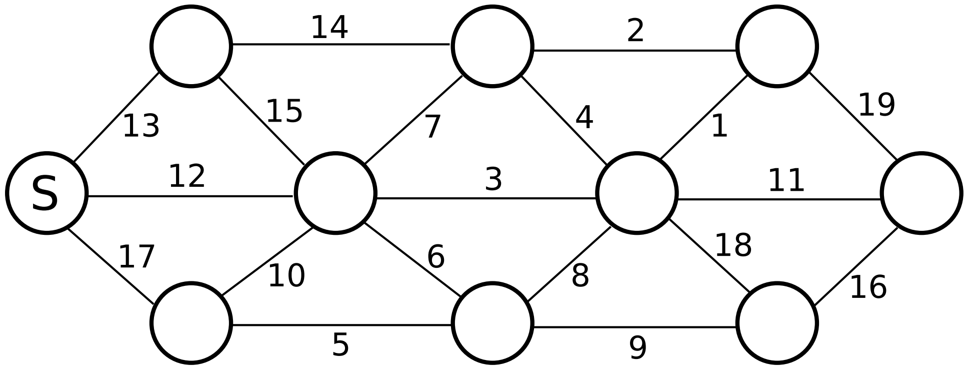


b) Now mark all the edges that are added by Dijkstra's algorithm, **and** in each vertex, write the distance of that vertex from the start vertex.

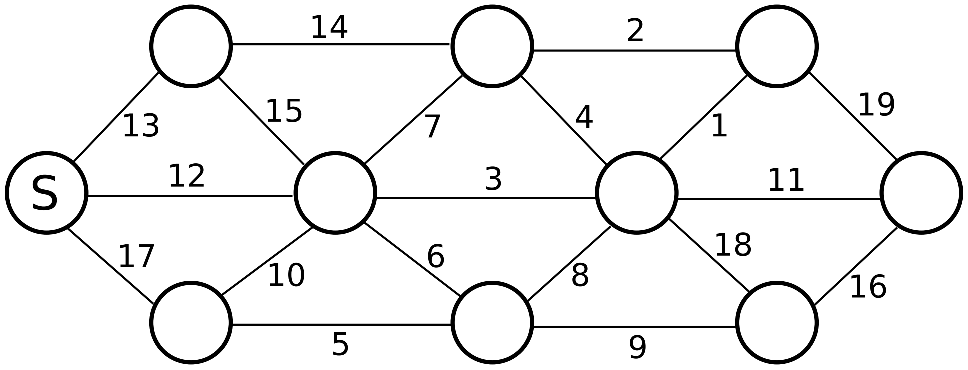


Problem 1 is continued on the back!

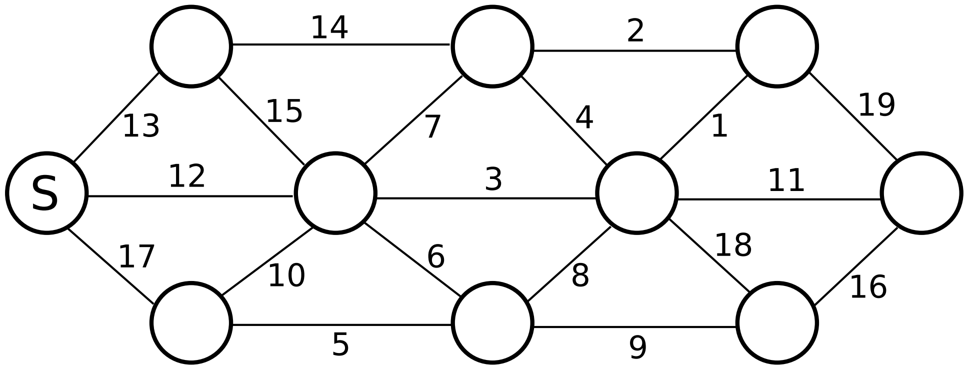
c) Suppose that **Kruskal's algorithm** is applied to find the minimal spanning tree for the following graph. Mark the **first six** edges that are added to the graph by Kruskal's algorithm.



d) Suppose that **Prims's algorithm** is applied to find the minimal spanning tree for the following graph, stating from the vertex on the left (marked with an "S"). Mark the **first six** edges that are added to the graph by Prims's algorithm.



e) Now, mark all the edges that are part of the minimal spanning tree for the graph (using any method).



2. When applying binary search to a sorted array of n items, you look at the $\frac{n}{2}$-th item to decide whether to continue the search in the first half or in the second half of the array. Here is an implementation of binary search, copied from the first in-class test:

```
int binsearch(double[] A, double x) {
    int low = 0;
    int high = A.length-1;
    while ( low < high ) {
        int mid = (low + high) / 2; // Note: low <= mid < high
        if (x <= A[mid])
            high = mid;
        else // x > A[mid]
            low = mid+1;
    }
    if (low == high && A[low] == x)
        return low;
    else
        return -1;
}
```

Consider a similar *ternary search* in which you look at the $\frac{n}{3}$-th and (possibly) the $\frac{2n}{3}$-th items in a sub-array to decide whether to continue the search in the first third, second third, or third third of the sub-array.

a) Write a Java method that implements this ternary search, for an array of type `double[]`. (Note that the computation "$mid = (high + low)/2$" could also be written "$mid = low + (high - low)/2$", and that the second formula generalizes more easily to ternary search.)

b) How many comparisons are done, on average, when using ternary search on an array of size n? Explain. (Look for an actual exact formula, not just a "Big-Theta" analysis.)

c) Is ternary search an improvement on binary search? Why or why not? (If you don't want to do the math in general, you can look at a specific array size, say $n = 1000$. But you can get a more general answer by using the basic log identity $\log_b(a) = \frac{\ln(b)}{\ln(a)}$.)

3. In a *set*, order does not matter and duplicates are not allowed. In a *multiset* (also called a *bag*), duplicates **are** allowed, but order still does not matter. An element of a multiset has a *multiplicity*, which tells how many times it occurs in the multiset. Operations on a multiset, m, include: $m.add(x)$ for adding the item x to m; $m.multiplicity(x)$ for getting the multiplicity of x in m (if x does not occur in m at all, then its multiplicity is zero); and $m.size()$ for determining the number of items in m, counting multiplicities (so that if m consists of two copies of a and three copies of b, then $m.size()$ is 5).

Consider the following three implementations for a multiset of *Strings* in Java. For each implementation, outline an implementation of the three operations $m.add(x)$, $m.multiplicity(x)$, and $m.size()$, and briefly discuss the run time of the implementation.

a) An unsorted *ArrayList<String>*
b) A sorted *ArrayList<String>*
c) A *HashMap<String,Integer>*

There are two more problems on the back!

4. This problem concerns distance between vertices in an unweighted graph. In an unweighted graph, the length of a path is the number of edges in that path, and the **distance** from one vertex v to another vertex w is the length of the shortest path from v to w. Give clear pseudocode algorithms to solve each of the following problems. (Hint: Use breadth-first search.)

a) Given a starting vertex, v, print out every vertex whose distance from v is less than or equal to 5.

b) Given a vertex v, find the largest distance from v to any other vertex in the graph. That is, find a vertex w which can be reached from v, and whose distance from v is as large as possible, and return the distance of that w from v

5. A graph can be represented either as an adjacency matrix or as an array of adjacency lists. For a given graph, the two representations will use different amounts of memory. For very sparse graphs, adjacency lists will use less memory. For graphs that are nearly complete, an adjacency matrix will use less memory. But for a particular graph, the answer will depend on the number of edges in that graph as well as on the details of how the matrix and lists are implemented.

Investigate the following question, and write a report on your results: Choose two particular, explicit representations for **directed graphs** in Java, one representing graphs using adjacency matrices and one using adjacency lists. For example, for the adjacency martix representation, you could use a two-dimensional array of *boolean*, and for the adjacency list representation, you could use an array of pointers to simple linked lists. Given a graph with n vertices and e edges, which representation will use less memory? The answer depends pretty much on how big e is compared to n^2, where n^2 is the maximum possible number of edges. Your answer will be along the lines of: The adjacency matrix takes less memory if e is greater than such-and-such a percentage of n^2. Your assignment is to produce some estimate of the cutoff percentage. Since the percentage depends to some extent on n, your answer will be a rough estimate rather than an exact percentage. (But you must, of course, show your work and explain your answer.)

For your computation, you can assume: A *boolean* value in Java occupies one byte of memory. A Java object uses 8 bytes, in addition to the memory for the member variables that it contains. An *int* uses 4 bytes. And a pointer uses 8 bytes.