Last name _____

First name _____

LARSON—OPER 635–SAGE WORKSHEET 13
Spanning Tree TSP Tour

1. Log in to your Sage Cloud account.

   (a) Start Firefox or Chrome browser.

   (b) Go to `http://cloud.sagemath.com`

   (c) Click "Sign In".

   (d) Click project **Classroom Worksheets**.

   (e) Click "New", call it **s13**, then click "Sage Worksheet".

   The degree relaxation TSP produced a "tour" that had only some of the features of a tour. Among other things it included sub-tours (or islands). This provided a lower bound on the length of an optimal tour.

   Now we will find a tour that satisfies the degree relaxation constraints **and** that is a genuine tour. We will start with a minimum weight spanning tree and use that as the basis for our construction. The length of this tour will be an upper bound for the length of an optimal tour.

   A **spanning tree** of a connected graph g is a spanning subgraph of g that is a tree. So, if the order of g is n, a spanning tree will have $n-1$ arcs.

2. Draw a graph g with 4 nodes labeled 0,1,2,3 and with weights/costs/distances equal to the sum of the node labels. Find 2 different spanning trees for g.

3. For a weighted graph, a **minimum weight spanning tree** is a spanning tree where the sum of the weights of the arcs is at least as small as the sum of the weights of the arcs for any other spanning tree. Find a minimum weight spanning tree for g.

   Kruskal's algorithm for finding a minimum weight spanning tree for a connected weighted graph consists in repeatedly choosing an arc of minimum weight that does not create a cycle together with the already-chosen arcs. This algorithm is implemented in Sage.

4. Write code for graph g or cut-and-paste it from an earlier worksheet. Then evaluate: `g.weighted(True)` to tell Sage to think of g as a *weighted* graph (not just a *labeled* graph); this is necessary for Sage's spanning tree function.

5. Evaluate: `g.min_spanning_tree()`.

6. This list of weighted edges can be turned into a graph.
   Evaluate: `spanning_tree_g = Graph(g.min_spanning_tree())`. To view this graph, evaluate: `spanning_tree_g.show()`.

7. To find the sum of the weights of this spanning tree, evaluate: `sum([c for (a,b,c) in g.edges()])`. This is necessarily a lower bound for the minimum tour length for our salesman (why?).

8. Cut-and-paste your code from the previous worksheet to re-create the graph DFJ (of the 42 US cities and their distances). Evaluate: `DFJ.weighted(True)` to make this a weighted graph. Now find a minimum spanning tree, and draw it on a map.

9. Find the sum of the weights of this spanning tree.

10. Now we will write a depth-first search algorithm (this happens in the 2nd while loop) that traces a tour around a spanning tree, searches the current vertex for a neighbor that it can go to and, if none exists, backtracks to previously visted nodes until it finds one that does. (Since this produces a genuine tour, the result is an upper bound on an optimal tour length—in fact, it can be proved, it os no more than double the length of an optimal tour). Evaluate:

```
def spanning_tree_tour(g):
    n = g.order()
    V = g.vertices()
    g.weighted(True)
    t = Graph(g.min_spanning_tree())

    current = 0
    tour = [0]
    while len(tour) < n:
        tour_copy = copy(tour)
        while True:
            test = tour_copy.pop()
            N = t.neighbors(test)
            accessible = [v for v in N if v not in tour]
            if len(accessible) > 0:
                current = accessible.pop()
                break
        tour.append(current)

    length = g.edge_label(0,current)
    length += sum([g.edge_label(tour[i],tour[i+1]) for i in range(n-1)])
    return length, tour
```

11. Use `spanning_tree_tour()` to find a tour for g. What is the length? Draw tour over your spanning tree—use a different color.

12. Use `spanning_tree_tour()` to find a tour for DFJ. What is the length? Draw the spanning tree and the tour in different colors.