1. (a) Start Chrome browser.
    (b) Go to \texttt{http://cocalc.com} and “Sign In”.
    (c) Click project \texttt{Math 356}.
    (d) Click “New”, call it \texttt{c32}, then click “Sage Worksheet”.

A spanning tree of a connected graph $G$ is a subgraph of $G$ that contains all the vertices of $G$ which is a tree. For a connected weighted graph, a minimum weight spanning tree is one whose edges have a minimum weight sum for all possible spanning trees.

2. Let \texttt{pete} be the Petersen graph. Evaluate \texttt{pete=graphs.PetersenGraph()} and “show” the graph.

We need some examples of weighted graphs. We previously coded the following function that puts random integer weights on the edges of a given graph $g$:

\begin{verbatim}
def random_integer_edge_weights(g,n): #weights will be in the interval $[1..n]$ 
    E = g.edges()
    for e in E:
        random_weight = randint(1,n)
        g.add_edge(e[0],e[1], random_weight)
    return g
\end{verbatim}

3. Let \texttt{pete1=random_integer_edge_weights(pete,10).show(edge_labels=True)}. “Show” it and draw what you get. (This will be different for each student).

4. Code the following improved implementation of Kruskal’s algorithm.

\begin{verbatim}
def kruskal2(g): #assumes g has real-number edge weights 
    E = g.edges()
    E_sorted = sorted(E, key = lambda x: x[2])
    T_edges = []
    for e in E_sorted:
        temp = copy(T_edges)
        temp.append(e)
        if g.subgraph(edges=temp).is_forest():
            T_edges.append(e)
    return g.subgraph(edges=T_edges)
\end{verbatim}
5. kruskal2 returns a graph (actually a tree). Let’s try it on pete1. Evaluate: `pete_tree=kruskal2(pete1)` Then show it to see the result. Remember to show the edge labels too!

Our next algorithmic problem is to find a shortest weighted path between a pair of vertices in a graph. We discussed the Dijkstra’s algorithm pseudo-code below (from our book). Let’s implement it. Instead of using lists—which depend on an ordering of the vertices—let’s use dictionaries. Different data structures have advantages and disadvantages—it’s less natural to iterate over dictionaries—but we can access their entries by their keys very efficiently.

---

**Algorithm 3.5 A general template for Dijkstra’s algorithm.**

**Input** An undirected or directed graph \( G = (V, E) \) that is weighted and has no self-loops. The order of \( G \) is \( n > 0 \). A vertex \( s \in V \) from which to start the search.

**Vertices are numbered from 1 to \( n \), i.e. \( V = \{1, 2, \ldots, n\} \).**

**Output** A list \( D \) of distances such that \( D[v] \) is the distance of a shortest path from \( s \) to \( v \). A list \( P \) of vertex parents such that \( P[v] \) is the parent of \( v \), i.e. \( v \) is adjacent from \( P[v] \).

1. \( D \leftarrow [\infty, \infty, \ldots, \infty] \) \( \triangleright \) \( n \) copies of \( \infty \)
2. \( D[s] \leftarrow 0 \)
3. \( P \leftarrow [] \)
4. \( Q \leftarrow V \) \( \triangleright \) list of nodes to visit
5. while \( \text{length}(Q) > 0 \) do
6. find \( v \in Q \) such that \( D[v] \) is minimal
7. \( Q \leftarrow \text{remove}(Q, v) \)
8. for each \( u \in \text{adj}(v) \cap Q \) do
9. \( \text{if } D[u] > D[v] + w(vu) \) then
10. \( D[u] \leftarrow D[v] + w(vu) \)
11. \( P[u] \leftarrow v \)
12. return \((D, P)\)

---

6. Evaluate \( D=\{"A":5, "B":\infty\} \). \( D \) is a Python dictionary. The strings “A” and “B” are keys, while 5 and \( \infty \) are values. We can access the values with the keys. Evaluate \( D["A"] \) and \( D["B"] \). You can change the value of a dictionary entry by assignment: to change the value associated to “A” to 4, use \( D["A"]=4 \). That’s it. To see that it changed evaluate \( D["A"] \). To see the whole dictionary just evaluate \( D \).

7. We’ll see that its natural to use dictionaries for \( D \) and \( P \) (even the pseudo-code suggests it). For \( Q \) lists are probably more natural (Python lists even have a .remove() method!). The ingredients we’ll need are \( \text{len}(Q) \) for “length(Q)” in the pseudo-code, \( g.\text{neighbors}(v) \) for the neighbors of \( v \) in graph \( g \) (that’s “adj(v)” in the pseudo-code). To get a list of vertices that are neighbors of \( v \) and in \( Q \), use \([w \text{ for } w \text{ in } g.\text{neighbors}(v) \text{ if } w \text{ in } Q]\). To get the weight of an edge \( e \) that the last element in the triples representing the edge—and since Python indexing starts at 0, we need \( e[2] \).

8. Try to hack out an implementation of Dijkstra’s algorithm using the above pseudo-code and these ingredients. Remember those “←” symbols are assignment operators—that’s just “=” in Python.