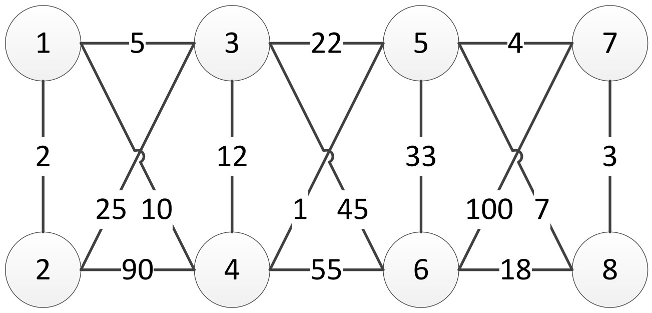
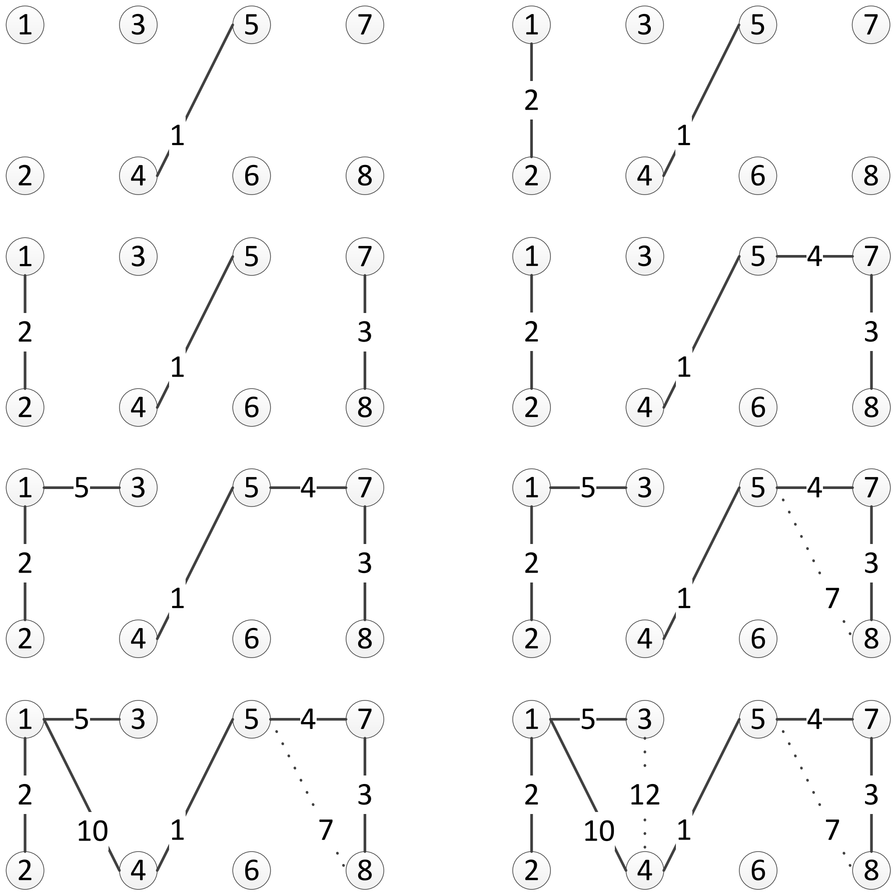
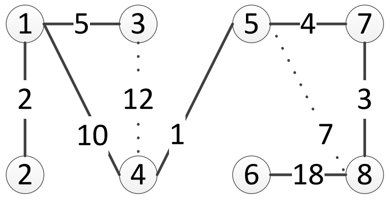
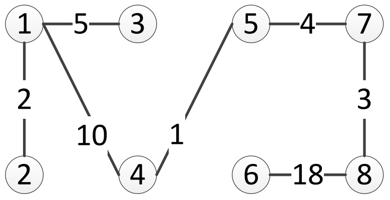
R-7.7

Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Kruskal's algorithm on this graph. (Note that there is only one minimum spanning tree for this graph.)





C-7.3

Consider the following greedy strategy for finding a shortest path from vertex start to vertex goal in a given connected graph.

1: Initialize path to start.

2: Initialize VisitedVertices to {start}.

3: If start=goal, return path and exit. Otherwise, continue.

4: Find the edge (start, v) of minimum weight such that v is adjacent to start and v is not in VisitedVertices.

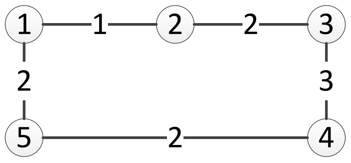
5: Add v to path.

6: Add v to VisitedVertices.

7: Set start equal to v and go to step 3.

Does this greedy strategy always find a shortest path from start to goal? Either explain intuitively why it works, or give a counter example.

This greedy strategy cannot always find the shortest path from start to goal. Consider the following counter example:



Assume that we want to find a shortest path from 1 to 4. If we follow this greedy strategy, we will select the path between 1 and 2 which is the shortest edge from 1. And next we will continue to select the edge between 2 and 3 and the edge between 3 and 4. Then, we will get the path (1->2->3->4) whose length is 6. However, the true shortest path is (1->5->4), which the length is 4. As a result, this greedy strategy cannot find the shortest path.

C-7.7

Suppose you are given a diagram of a telephone network, which is a graph G whose vertices represent switching centers, and whose edges represent communication lines between two centers. The edges are marked by their bandwidth. The bandwidth of a path is the bandwidth of its lowest bandwidth edge. Give an algorithm that, given a diagram and two switching centers a and b, will output the maximum bandwidth of a path between a and b.

We can design an algorithm like Dijkstra. We define that b[v] represents the bandwidth between a and v. For each b[v], we initialize them to 0, except for the b[a], which we initialize to infinity. And we also define two sets S1 and S2. For S1, we initialize it to Ø, and for S2, we initialize it to all vertices in V. And w(v, u) will represent the bandwidth between v and u. First, we will find the max b[v] in S1 and add it to S2. Next, we will find all vertices that have edge with vertex v. For all these vertices, if min{b[v], w(v, u)} is bigger than b[u], then will update b[u] to min{b[v], w(v, u)}.

Like the Dijkstra algorithm, assuming n vertices and m edges in the graph, the running time is $O((n+m)\log n)$. Inserting all vertices in a heap uses $O(n \log n)$ time. And for the while loop, it uses $O(\log n)$ time to remove vertex v from S2 to S1, and $O(dev(v)\log n)$ time to perform the relaxation procedure on these edges.

```
Algorithm bandwidth(G, a, b):
 b[a] ← ∞
 for each vertex v in V do
   b[v] ← 0
 S1 ← Ø
 S2 ← V
 while S2 ≠ Ø do
   v ← the vertex which has the max bandwidth
   S1 ← S2 ∪ {v}
   for all vertices u which have edge with v do
     if min{b[v], w(v, u)} > b[u] do
       b[u] ← min{b[v], w(v, u)}
 return b[b]
```

C-7.8

NASA wants to link n stations spread over the country using communication channels. Each pair of stations has a different bandwidth available, which is known a priori. NASA wants to select n-1 channels (the minimum possible) in such a way that all the stations are linked by the channels and the total bandwidth (defined as the sum of the individual bandwidths of the channels) is maximum. Give an efficient algorithm for this problem and determine its worst-case time complexity. Consider the weighted graph G=(V, E), where V is the set of stations and E is the set of channels between the stations. Define the weight w(e) of an edge e∈E as the bandwidth of the corresponding channel.

We can design an algorithm like Kruskal. For each step, we pick each edge which has the greatest weight.

At first, each vertex is in its own cluster all by itself. Next, we consider each edge in turn, ordered by decreasing weight. If an edge connects two different clusters, this edge will be added to the set of edges of the spanning tree, and the two clusters connected by this edge will be merged into a single cluster. If, on the other hand, this edge connects two vertices that are already in the same cluster, then this edge will be discarded. Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the final result.

Due to the theorem for Kruskal, the running time is $O(m \log n)$ which n is the number of station and m is the number of edge. In this algorithm, the worst-case time is to consider all the edges in this graph. The number of these edges is $n \times (n-1)$. As a result, the running time is $O(n^2 \log n)$.