1. **(5 points)** Compare what happens in Kruskal's algorithm versus Prim's algorithm if we run them on a graph that is *not* connected.

   Solution: When running Kruskal's algorithm on a disconnected graph a spanning forest will be returned, that is, the result will be a collection spanning trees, each of which is a MCST for each connected component in the original, disconnected, graph. This is the case because Kruskal's considers edges and will only add an edge if it does not create a cycle. Since the whole set of edges is ordered by cost, it follows that the subset of edges for each connected component will also be ordered by cost, and considering there is no case where a cycle can be created between connected components, it can be seen that an MCST will be created for each connected component. A full proof of this can be seen in problem 2.9 solution in the text.

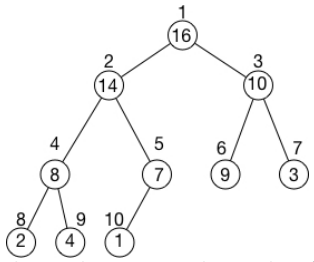
   Prim's algorithm, on the other hand, will not create a spanning forest. Strictly speaking, the algorithm will fail when running on a disconnected graph. This is the case because Prim's algorithm works by initially selecting an arbitrary vertex then building by adding edges that are reachable from that vertex, however it will never be able to reach other connected components. The loop will never terminate because B will never be equal to V and this will inevitably lead to a problem in line 4, there will be a case where there does not exist an edge with one vertex in B and one not in B. The exact details of what will happen will be implementation specific, but at the time of failure B will be a MCST for one, arbitrary, connected component.

2. **(5 points)** Adapt Prim's algorithm to graphs that may include edges of negative costs; give an example of an application where negative costs may occur naturally.

   Solution: Prim's algorithm is naturally able to handle both negative and positive weights without issue. One way to see this would be to add an arbitrarily large constant to all the weights before processing to make them all positive then subtracting this from the edges in the final MCST. It can be seen that the same edges will be picked regardless of if this constant is added or not, and the same MCST will be generated.

   One example of a case where negative weights naturally occur is taxi drivers. Driving a passenger to a destination can be viewed as a positive cost and driving to a passenger can be viewed as a negative cost.

3. **(10 points)** A *binary heap* data structure is an array that we can view naturally as a nearly complete binary tree. Each node of the tree corresponds to an element in the array, as shown below:



16	14	10	8	7	9	3	2	4	1

note that the array has the following interesting structure: the parent of i is $\lfloor i/2 \rfloor$ and the left child of i is $2i$ and the right child of i is $2i + 1$.

With the binary heap data structure we can implement line 4. efficiently, that is, find the cheapest e. To this end, implement a min-priority queue B, which, during the execution of Prim's algorithm, keeps track of all the vertices that are *not* in the tree T. The min-priority queue B uses the following key attribute: minimum weight of any edge connecting the vertex to T (and ∞ if no such edge exists).

Describe the details of the above scheme, and implement it in Python 3.

Solution: An example Haskell implementation can be seen in `a2-sol-q3-comp554-w18.hs` and a Python 3 Implementation can be seen in `a2-sol-q3-comp554-w18.py`.