1. (8 Pts) Analyzing a Minimum Spanning Forest Algorithm

```
Build-MSF-By-Add-And-Fix (G, w)
1 F = {} // empty set
2 for each edge e ∈ E, taken in arbitrary order
3    F = F ∪ {e}
4    if F has a cycle c
5       let e' be a maximum weight edge on c
6       F = F − {e'}
7 return F
```

a. Complete the following proof that this algorithm constructs an MSF. A partial proof is given below; you complete it.

You will use this counterpart to the safe edge theorem, which we prove for you:

Safe edge removal theorum: Given a graph with a single cycle, removing the edge with the maximum weight on that cycle results in a forest with the smallest total weight.

Proof:
Since there is only one cycle, to create a forest, we need to remove only one edge from the cycle. Removing any edge that is not the largest from the cycle will result in a forest with a larger total weight. Q.E.D.

You will now prove the rest by filling in the following:

Claim: When Build-MSF-By-Add-And-Fix terminates, it will produce a Minimum Spanning Forest.

Proof: By induction on the number of edges inspected of this proposition:
Proposition P(k): "After k edges have been inspected (after k iterations), the graph F is a minimum spanning forest (MSF) of the graph G that contains only the edges inspected in line 2."

Base k = 1:
After an edge is inspected, it will be the only edge that will have a cycle, resulting in creating a Minimum Spanning Tree.

Induction P(k) => P(k+1) (Assume P(k) is true and prove P(k + 1) is true.) There will be two cases to consider:

Case 1: The addition of an edge creates a cycle.
In the case of the inductive statement, and the assumption that P(k) is true, begin by adding the P(k + 1) edge. This edge, denoted as e creates a cycle. Proposition P(k) states that the graph has a Minimum Spanning Forest (MSF) that contain only the edges that are inspected in line 2. This means that if and when the cycle is created by adding edge e from P(k + 1),

the Safe Edge Removal Theorem can be implemented, stating that the maximum weight in the cycle is a result of the creation of the Minimum Spanning Tree.

Case 2: The addition of an edge does not create a cycle.
In the case of the inductive statement and the assumption that P(k) is true, begin by adding the P(k + 1) edge. This edge, denoted as e will not create a cycle. This results in a a forest, but there will be a minimum weight over the aggregation of edges due to the creation of two separate trees. By definition, this still results in a forest, meaning it is a minimum spanning forest.

Conclusion: Use the above to conclude that F is correct when the loop terminates and the algorithm returns F:

b. Describe an efficient algorithm for finding the maximum weight edge on a cycle, to use in lines 4 - 5. Give the time complexity.

An efficient algorithm for finding the maximum weight edge on a cycle can be implemented in lines 4 - 5 of this algorithm. Currently, line 4 checks whether a forest has a cycle, and if so, allows an edge to have a maximum edge on that given cycle. An algorithm that can be used to detect this is to have a variable keep track of these maximum weight edges and compare them to each other. Once the maximum edge in the cycle is found, this value can be returned. The time complexity for recording the actual weight on the cycle can run in O(1) time, but in regards to cycling through the data that stores the edge values would run in O(n) time, with n being the number of edge weights discovered.

c. Analyze the time complexity of Build-MSF-By-Add-And-Fix, assuming you use the algorithm in (b):

Assuming the implementation of the algorithm designed in b, which keeps track of the weights of the edges, the runtime as a result of this addition to Build-MSF-By-Add-And-Fix is $O(V + E)$. Although stated above that the implemented algorithm for finding a max weight on the edges is O(n), it is known that each edge weight accounts for a given edge in the cycle. Therefore, the edges can be discussed as E in the runtime analysis. So the final time complexity of this algorithm with the addition of finding the maximum weight edge on a cycle is $O(V + E)$.

2. (6 pts) MST on Restricted Range of Integer Weights

Suppose edge weights are restricted as follows. For each restriction:
• Describe a modification to Kruskal's algorithm that takes advantage of this restriction, and
• Analyze its runtime to prove that your modified version runs faster.

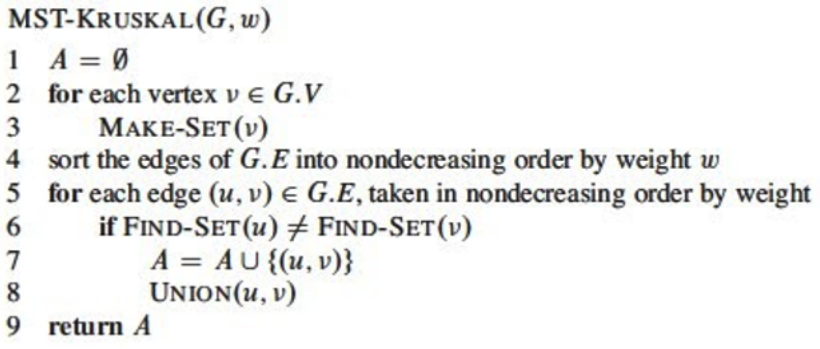
a. All edge weights are integers in the range from 1 to C for some constant C.
In this version of Kruskal's algorithm, all edge weights are integers in the range 1 to C for

some given constant C. A modification of this algorithm that takes advantage of this given restriction is illustrated in line 4 in the algorithm, which sorts the edges of G.E. into nondecreasing order by weight w. Given the weights of the integers are in the range 1 to C, an implementation of Counting Sort would order these weighted values, with a runtime of O(E), where E is the amount of edges in the weighted graph. Since Counting Sort is taken into account in this new algorithm, the overall run time is $O(C + E + VlogV)$, which is calculating the constant C, and the edges and vertices.

b. All edge weights are integers in the range from 1 to |V|.
In this version of Kruskal's algorithm, all edge weights are integers in the range from 1 to |V|. A modification of this algorithm that takes advantage of this given restriction can again be illustrated in line 4 of the algorithm, which sorts the edges of G.E. into nondecreasing order by weight w. By implementing Counting-Sort, the edges can be sorted with a $O(V + E)$ runtime. This modified version of Kruskal's algorithm can now run in $O(E + VlogV)$ time., since this takes into account the number of edges and all the vertices it must travese.

3. (10 pts) Suppose we change the representation of edges from adjacency lists to matrices.

Assume that vertices are represented as integers [1...|V|] and that G.E. is a symmetric adjacency matrix where each G.E[u,v] contains the weight on edge (u,v) ∈ E, or 0 if there is no edge.

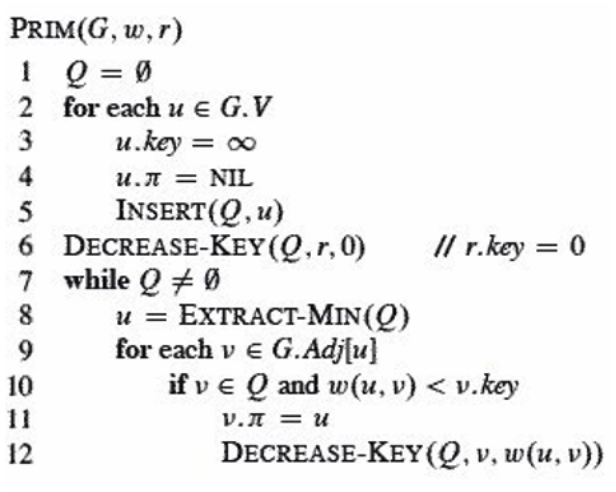


```
MST-KRUSKAL($G, w$)
1  $A = \emptyset$
2  for each vertex $v \in G.V$
3      MAKE-SET($v$)
4  sort the edges of $G.E$ into nondecreasing order by weight $w$
5  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6      if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          $A = A \cup \{(u, v)\}$
8          UNION($u, v$)
9  return $A$
```

a. What line(s) would have to change in the CLRS version of Kruskal's algorithm, (shown), and how? (No need to write the code; just explain the changes required for the matrix representation.)

A change that would be required for the given matrix representation is an adjustment to line 4 in this code. Currently, line 4 will sort the edges of G.E into nondecreasing order by weight w. Since it currently sorts the edges, the adjustment would need to take into account the set of edges within the matrix, and scan through the given edges, sorting them accordingly.

b. What would be the resulting asymptotic runtime of Kruskal's algorithm on both dense ($E = O(V^2)$) and sparse ($E = O(V)$) graphs, and why?
The asymptotic runtime, respective to dense graphs is $O(V^2 logE)$, and the runtime respective to spare graphs would be $O(V^2)$. For a dense graph it is not possible to have a combination of V^2 edges, while in a spare graph, the sorting of an edge will run in $O(ElogE)$, where V^2 will dominate. This means that the resulting asymptotic runtime of Kruskal's algorithm is $O(V^2)$.



```
PRIM(G, w, r)
1  Q = ∅
2  for each u ∈ G.V
3      u.key = ∞
4      u.π = NIL
5      INSERT(Q, u)
6  DECREASE-KEY(Q, r, 0)      // r.key = 0
7  while Q ≠ ∅
8      u = EXTRACT-MIN(Q)
9      for each v ∈ G.Adj[u]
10         if v ∈ Q and w(u, v) < v.key
11             v.π = u
12             DECREASE-KEY(Q, v, w(u, v))
```

c. What line(s) would have to change in the CLRS version of Prim's algorithm (shown), and how?

The instruction in line 9 of Prim's algorithm currently analyzes the case for a vertex being an element of G.Adj[u]. This must be modified to instead accept the vertices and edges in terms of an adjacency matrix, where the cycling of this representation of data is searching from the point u. Another modification that follows is between lines 10 to 11, where currently, if the vertex is an element of a queue, and if the weight of the edge is less than the vertex's key, the the vertex parent is set to be u. This would need to be modified to take into account the matrix that was changed in line 9, where the representation of this matrix as an array would need to store the value of parent in another matrix or array.

d. What would be the resulting asymptotic runtime of Prim's algorithm on both dense ($E = O(V^2)$) and spare ($E = O(V)$) graphs, and why?

The asymptotic runtime, respective to dense graphs is $O(V^2 logV)$, and the runtime respective to sparse graphs would be $O(V^2 + V logV)$. This means that the resulting overall runtime of Prim's algorithm is $O(V^2 + ElogV)$.

4. (6 pts) Divide and Conquer MST

Consider the following divide-and-conquer algorithm for computing minimum spanning trees. The intuition is that we can divide a graph into half, solve the MST problem for each half, and then find a minimum cost edge spanning the two halves. More formally:

Given G = (V,E), partition V into V_1 and V_2 such that $_1|$ and $_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1 and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve the MST problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2) and use this edge to join the resulting two minimum spanning trees into a single spanning tree.

Prove that this algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

This algorithm can correctly compute a minimum spanning tree. An example for this can be produced through providing a counter example, where it shows why it will fail.

Suppose there is a given set of four vertices based on the example outlined above. These vertices can be denoted through letters, such as a, b, c, d. The partition of these vertices can be as such, where V_1 and V_2 differ by at the most 1, and E_1 can be represented as the set of edges that are incident only on vertices in V_1 and E_2 be the set of edges on incident in vertices V_2. If this is graphically represented, this can be cut with V_1 represented as the nodes a,b and V_2 being represented as c,d. If the given weight w(u,v) from a to be is smaller than the given weight w(u,v) from c to d, this representation can be mapped differently with different values because of this cut. Overall, this proves the counterexample, since the resulting MST, depending on the edge weights will be either lighter or heavier than the other.