An algorithm is said to run in polynomial time if it runs in time $O(n^d)$ for some constant $d$ on inputs of size $n$. Each problem is worth 10 points:

1. Show an execution of Kruskal’s algorithm to compute the minimum spanning tree of the following graph. Show the state of the connected components (union find) data structure at each step:

![Graph Diagram]

Solution. First, sorts all edges by their weights (breaking ties arbitrarily, so there might be different MST)

1 - $AB, DG, FG, GH$
2 - $BC, CG$
3 - $CD$
4 - $DH, AE$
5 - $EF$
6 - $BF, BG$
8 - $AF$

Second, loops over all edges $\{u, v\}$ one by one, if $u$ and $v$ are not in the same set, union them. If $u$ and $v$ are in the same set, don’t do anything.

Step 1: $Union(A, B)$
Step 2: $\text{Union}(D, G)$

Step 3: $\text{Union}(F, G)$

Step 4: $\text{Union}(G, H)$

Step 5: $\text{Union}(B, C)$
Step 6: $Union(C, G)$

Step 7: $Union(C, D)$, skip because $C$ and $D$ are already in the same set
Step 8: $Union(D, H)$, skip because $D$ and $H$ are already in the same set
Step 9: $Union(A, E)$
The final MST:

2. Prove or disprove the following: Given any undirected graph $G$ with weighted edges, and a minimum spanning tree $T$ for that $G$, there exists some sorting of the edge weights $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$, such that running Kruskal’s algorithm with that sorting produces the tree $T$.

Solution: We shall show that there is a sorted order producing the MST $T$. Choose a sorted order $w(e_1) \leq \ldots \leq w(e_m)$ satisfying the property that if $e_i, e_j$ are such that $w(e_i) = w(e_j)$ and $e_i$ belongs to the tree but $e_j$ does not, then $e_i$ comes before $e_j$. In other words, within the edges of the same weight, put the edges of the tree first.

We shall prove by induction on $i$ that at the $i$'th step, Kruskal’s algorithm will include $e_i$ if and only if $e_i$ belongs to $T$.

For the base case of $e_1$, observe that the algorithm must include $e_1$, and $e_1$ must belong to the tree. This is because if $e_1$ does not belong to $T$, then adding $e_1$ to $T$ gives a cycle, and all the other edges of the cycle must have larger weight than $e_1$. Then we delete one of the other edges from the cycle we obtain a new tree of lighter total weight, contradicting the fact that $T$ is an MST.

Now, for $i > 1$, let $K$ denote the edges picked by the algorithm so far. If $e_i$ belongs to the tree then Kruskal’s algorithm will include $e_i$, since this does not create any cycles in the edges $K$ (which all belong to $T$ by induction).

If $e_i$ does not belong to the tree, then we claim that $e_i$ must create a cycle among the edges of $K$, and so Kruskal’s algorithm will not include it. This is because $e_i$ must create a cycle in $T$, so if it does not create a cycle in $K$, then some edge of the cycle in $T$ does not belong to $K$, so this edge must be heavier than $e_i$. Adding $e_i$ to $T$ and deleting the heavier edge gives a cheaper tree, which is not possible since $T$ is an MST.

3. You are given a graph $G$ with $n$ vertices and $m$ edges, and a minimum spanning tree $T$ of the graph. Suppose one of the edge weights $w(e)$ of the graph is updated. Give an algorithm that runs in time $O(m)$ to test if $T$ still remains the minimum spanning tree of the graph. You may assume that all edge weights are distinct both before and after the update. HINT: If $e \in T$, consider the cut obtained by deleting $e$ from $T$. If $e \notin T$, consider the cycle formed by adding $e$ to $T$.

Solution. The pseudocode is given below.

Runtime Analysis: All graph traversals take $O(n)$ time, and looping over all edges takes $O(m)$. So the total time complexity is $O(n + m)$ which is $O(m)$ because $n$ is not greater than $m$. 

4-4
**Input:** Graph $G(V, E)$, MST $T$, edge $e \in E$, weight $w'$ that $e$ will be updated to

**Output:** Whether or not $T$ remains the MST of $G$ after weight$(e) = w'$

if $e = \{u, v\} \in T$ then
  Delete $e$ from $T$;
  Use depth first search in $T$ from $u$ to all reachable vertices and mark each vertex as being in $S$;
  for all edges $e'$ in $E$ do
    if $e$ crosses the cut $S$ and weight$(e') < w'$ then
      return False;
  end
  end
else
  Compute the path in $T$ from $u$ to $v$;
  if weight(the heaviest edge in the path) > $w'$ then
    return False;
  end
end
return True;

**Proof of Correctness:** There are two cases in this problem, and the algorithm solves each one separately.

First, when $e$ is in $T$, consider the cut $S$ obtained after removing $e$. By the cut property, the edge that connects the two cuts to make the MST needs to have the lightest weight among all other crossing edges. Thus, if weight$(e)$ is not the smallest, $T$ is no longer the MST.

Conversely, if weight$(e)$ is still the lightest weight edge, then we claim that $T$ must remain an MST. To see this, suppose $T'$ is an MST in the new graph of lighter weight. Then without loss of generality $T'$ must contain $e$, for if $T$ does not contain $e$, we can exchange $e$ for an edge of $T'$ (one that belongs to the cycle formed by adding $e$ to $T'$ to obtain another MST). Then we see that since $T'$ and $T$ both contain $e$, $T'$ must have had lighter weight than $T$ even before $e$'s weight was updated, which is a contradiction.

The second case is when $e \notin T$. For $T$ to still be the MST, $e$ needs to be a heaviest edge in the cycle forming by adding $e$ to $T$ (by the cycle property, the heaviest edge in any cycle is always not in the MST).

Conversely, if $e$ is a heaviest weight edge in such a cycle, then $T$ must remain an MST. To see this, suppose $T'$ is an MST. We can assume that $T'$ does not contain $e$, for if it does, we can exchange $e$ for an excluded edge from the cycle (which must be of the same weight as $e$, since $e$ is of heaviest weight). Since both $T$ and $T'$ do not contain $e$, if $T'$ has smaller weight than $T$ after the update, it must have had smaller weight before the update, which is a contradiction.

4. You are given two sorted lists of integers of length $m$ and $n$. Give an $O(\log m + \log n)$ time algorithm for computing the $k$’th smallest integer in the union of the lists.
Solution: We shall carry out a variant of binary search. Let \( x_1, \ldots, x_m \) and \( y_1, \ldots, y_n \) be the given lists.

(a) If \( m = 1 \)
   i. If \( y_{k-1} \leq x_1 \leq y_k \), output \( x_1 \),
   ii. Else If \( x_1 \leq y_k \), output \( y_{k-1} \)
   iii. Else output \( y_k \).

(b) If \( n = 1 \)
   i. If \( x_{k-1} \leq y_1 \leq x_k \), output \( y_1 \),
   ii. Else If \( y_1 \leq x_k \), output \( x_{k-1} \)
   iii. Else output \( x_k \).

(c) If \( \lfloor m/2 \rfloor + \lfloor n/2 \rfloor \geq k \) then
   i. If \( x_{\lfloor m/2 \rfloor} \leq y_{\lfloor n/2 \rfloor} \), then recursively find the \( k \)'th smallest number in \( x_1, \ldots, x_m \) and \( y_1, \ldots, y_{\lfloor n/2 \rfloor} \).
   ii. If \( x_{\lfloor m/2 \rfloor} > y_{\lfloor n/2 \rfloor} \), then recursively find the \( k \)'th smallest number in \( x_1, \ldots, x_{\lfloor m/2 \rfloor} \) and \( y_1, \ldots, y_{\lfloor n/2 \rfloor} \).

(d) If \( \lfloor m/2 \rfloor + \lfloor n/2 \rfloor < k \) then
   i. If \( x_{\lfloor m/2 \rfloor} \leq y_{\lfloor n/2 \rfloor} \), then recursively find the \( k \)'th smallest number in \( x_{\lfloor m/2 \rfloor+1}, \ldots, x_m \) and \( y_1, \ldots, y_{\lfloor n/2 \rfloor} \).
   ii. If \( x_{\lfloor m/2 \rfloor} > y_{\lfloor n/2 \rfloor} \), then recursively find the \( k \)'th smallest number in \( x_1, \ldots, x_{\lfloor m/2 \rfloor} \) and \( y_{\lfloor n/2 \rfloor+1}, \ldots, y_n \).

To efficiently implement the recursive calls, we only pass in the end-points of the new input intervals used, rather than copying the whole input into a new array.

To prove correctness, when \( n = 1 \) or \( m = 1 \), the algorithm uses the sorted lists to find the \( k \)'th smallest element in constant time. In the other cases, the algorithm always eliminates half of one list. In case (b), i, we must have that the \( k \)'th smallest number is at most \( y_{\lfloor n/2 \rfloor} \), since there are \( \lfloor m/2 \rfloor + \lfloor n/2 \rfloor \geq k \) numbers that are at most \( y_{\lfloor n/2 \rfloor} \). Thus, the recursive step will correctly find the \( k \)'th smallest number in the overall list. All the other cases hold for the same reason: in each case we eliminate one half of one of the lists.

There can be at most \( O(\log n) + O(\log m) \) recursive calls, because in each call one of the lists is halved.