1.1)

Consider the direct proof: An MST is a tree. By the tree definition, this means the MST of G is connected, acyclic, and contains n-1 edges. When we add the edge e to the MST, we know this creates a cycle because the resulting graph G' would have n edges and be fully connected.

By the cycle property of MSTs, we know that if a cycle exists in G', the heaviest edge weight in that cycle must not be included in its MST. Because adding e creates exactly one cycle in the MST of G, we only need to consider this single cycle. We will compare the edge e to the other edges in the cycle- if e is lighter than the heaviest edge *p* in the cycle, we replace *p* with *e* to produce a valid MST. Otherwise, when *e* is the heaviest edge in the cycle, we simply keep the original MST by not adding *e*. In the end, we have created a new, valid MST by only considering the edges in the cycle, which are at most n-1 edges. Therefore, this process will take O(n) time.

1.2A) ALGORITHM

Assume we defined the entire union-find data structure discussed in lecture, including parents, def union(x,y), and def find(x).

```
def stream_MST ( V ,  edge_stream ):
    Initialize:
    mst_edges = [] */ let mst_edges be an empty array that will later store all the edges
    included in the MST /*
    do:
    while ( edge_stream.has_next() ) do:
        (u,v,w) ←edge_stream.next_edge();
        */ check if there is a cycle/*
        If find(u) == find(v) do:
            Max_edge ←null;
            Max_weight←−∞;
            */ find max edge weight/edge in the cycle/*
            For (u',v',w') in mst_edges do:
                If (find(u')==find(u) AND find(v')==find(v)) do:
                    If w' > max_weight do:
                        max_weight = w';
                        max_edge =(u',v',w');
            */ only replace if better than the max/*
            If w < max_weight do:
                Mst_edges.delete (max_edge);
                union(u,v);
                mst_edges.append((u,v,w));
        else do:
        */ if we get here, adding edge e doesn't create a cycle, so it is added to MST/*
            union(u,v);
            mst_edges.append((u,v,w));
    */after while loop finishes, we'll have created a valid MST/*
    return mst_edges;
```

1.2B) PROOF OF CORRECTNESS

The algorithm checks each edge being added if it creates a cycle and properly adds or omits said edge from the MST based on the cycle property we outlined in 1.1. Using the union-find structure, we check if e creates a cycle properly with the check find(u) == find(v).

Given by 1.1 and the cycle property, we maintain MST validity by removing the heaviest edge weight in the given cycle. The lines [**For** (u',v',w') **in** mst_edges **do: If** (find(u')==find(u) AND find(v')==find(v)) **do**:] check all other edges in the cycle and the nested lines of code store the max weighted edge.

After determining the max weighted edge in the given cycle, the algorithm checks [**If** w < max_weight **do**:]. When this condition is met, the added edge should be in the MST because another edge in the cycle has a higher weight, so the algorithm upholds these properties. Otherwise, the edge merely shouldn't be added to the MST, and the algorithm ensures this as well.

Finally, in the condition that a cycle is not created by adding a given edge, the edge is simply added to the MST, which is upheld by the code in the "else do" statement.

All-in-all, the algorithm only adds edges that connect separate components or uphold the cycle properties otherwise. The algorithm will eventually run out of new edges and will eventually terminate while upholding MST properties, and is therefore correct.

1.2C) COMPLEXITY

TIME COMPLEXITY: To analyze time complexity, we look at each step of the algorithm.

(1) [**while (** edge_stream.has_next() **) do:**]The main loop to process to edge stream will run once per edge and therefore had complexity O(n).
(2) [**If** find(u) == find(v) **do:**] For every edge, the algorithm calls find() on both vertices. As observed in lecture, we know this takes O($\alpha(n)$) time.
(3) [**for** (u',v',w') **in** mst_edges **do:**] For every other edge of n-1 edges, we compare with the current edge and run the line- [**if** find(u')==find(u) AND find(v')==find(v) **do:**]. In total, this could take O(n $\alpha(n)$)) time.
(4) [union(u,v);] when [w<max_weight]→ Replacing the edge with the union operation will also take O($\alpha(n)$) time as observed in lecture.
(5) If there is no cycle, we still run the union operation, which as observed has time complexity O($\alpha(n)$).

All other operations run in constant time. Therefore, we analyze the largest term of the combined complexity and conclude that **O(n* $\boldsymbol{\alpha(n)}$)** is the total runtime.

SPACE COMPLEXITY: To analyze space complexity, we look at the inputs, outputs, and temporary structures. Vertices and edge_stream take O(2n) space. Additionally, as observed in lecture, union-find and takes total O(2n) space by using two hash tables. Finally, the output of our algorithm- mst_edges- will take at most O(n) space as well. In total, the algorithm takes O(5n) space, which we denote as O(n) since we ignore leading constants.

2.2A)

The assertion that this greedy approach will read $\Omega(n)$ cells in the worst case is true. Consider the proof by contradiction:

Say this approach doesn't have an upper bound of $\Omega(n)$ cell reads. Let a be an array that reads as [1,2,3,4,5,6,7,8,9,23]. Let the randomly selected index position be 0. In this case, the greedy approach outlined would read cell 0: it will see if 1 is a peak, determine its neighbor(s) are greater, and then move on to its largest neighbor at index 1. This process will continue to the next cells until it reaches the last one at index 9, where we find a peak. This is an example of the worst-case scenario of this approach, and the algorithm took $\Omega(n)$ cell reads since it considered all nine cells, which contradicts my assertion that this approach does not have an upper bound of. By contradiction, we prove the assertion that $\Omega(n)$ is a lower bound of the outlined greedy approach to be true.

2.2B-1) ALGORITHM

```
def find_peak ( array ):
    */check in the beginning just to improve best case/*
    if len(array) ==0:
        throw illegal argument exception
    elif len(array) ==1:
        return array[0];

    */call recursive function/*
    return binary_peak (array, 0, len(array)-1);

def binary_peak ( array, left, right):
    middle = (left + right)//2;

    */base case/*
    if [ (middle ==0  OR array[middle]>array[middle-1]) AND (middle ==len(array)-1 OR
array[middle]>array[middle+1])] do:
        return middle;

    */recursively search left subtree/*
    elif (middle >0 AND array[middle]<array[middle-1]) do:
        return binary_peak(array, left, middle-1);

    else:
    */same for right subtree/*
        return binary_peak (array, middle+1, right);
```

2.2B-2) PROOF OF CORRECTNESS

Assertion 1: At each step the algorithm ensures that it's searching in a region of the array that contains a peak. The algorithm checks the middle item to see if it's a peak. If the middle is not a peak, the algorithm recursively checks the subarray with a greater neighbor. Then, it performs the same operation on said subtree's middle element. The algorithm will continue to recurse until a peak is found.

Assertion 2: The algorithm always finds a peak and terminates after. Given by the problem, every element is unique in the array; therefore, a peak must exist in the array. The algorithm properly traverses the subarrays in a manner that finds peaks and terminates after that peak is found. Therefore, a peak is found and the algorithm terminates.

2.2B-3) COMPLEXITY

TIME COMPLEXITY: To analyze time complexity, we examine every step of the algorithm. The algorithm follows a divide and conquer approach similar to a Binary search. Since every subarray is divided in half just like a binary search algorithm, we know the most amount of recursive calls to an input of size n will be $\log_2 n$. Additionally, work done in each recusive step is constant. Consequently, we denote overall time complexity as **O(log n),** since we disregard the base of log operators.

SPACE COMPLEXITY: To analyze space complexity, we look at recursive stack depth and other structures. As analyzed in the time complexity section, we determined the recursive depth to be O(log n). Additionally, the array will take O(n) space in the worst case. All-in-all, the space can be defined as O(n + log n), which we denote as **O(log n).**