Problems

1. Minimum Spanning Tree

Main Idea: We consider the two trees generating by removing the edge whose weight has changed from w to w'. If the new tree with w' is indeed a MST, the value of w' will be the smallest possible edge between the two trees.

Pseudocode:

```
MST(){
        Maintain Grouping[n] to store the group of each n nodes (either 1 or 2)
        Run DFS on u with the edges of T - (u,v)
                At each node v discovered, Grouping[v] = 1
        Run DFs on v with the edges of T - (u,v)
                At each node w discovered, Grouping[w] = 2

        Maintain a variable minimum, initialized to infinity
        For each edge m = (y,z) in G
                If Grouping[y] != Grouping[z]
                        minimum = min(minimum, m)
        Return minimum == w'
}
```

Proof of correctness:
We are given a MST with one edge that is changed. We split by the edge with weight w'. Since removing an edge of a MST disconnects the tree into two components, we can discover the nodes in each component and mark them accordingly. By the definition of a MST, the edge that connects the tree must have a minimum value among all edges between the two non-connected components. We loop through all the vertices and check each edge if it (1) links the two components and (2) is a smaller value than the previously found minimum.

Assume that the minimum edge chosen is not part of the MST. We know that some edge must be chosen between the two components or the graph will not be connected and by definition, not a MST. We also know that in order to be a MST, the edge with least cost must be chosen, which is what the algorithm above does. Therefore, we know that the minimum cost edge discovered must be equal to w' if T is still a MST.

Complexity:
Runtime: The two runs of BFS will take $O(\mathrm{m})$ time. Then, iterating through each edge to find the minimum value between the two components takes $O(\mathrm{m})$ time. Therefore, the overall time complexity is $O(\mathrm{m})$.
Space: The additional space required is the array Grouping, which will take an additioanl $O(\mathrm{n})$ space. Other variables used are stored in constant space, so we have an overall space complexity of $O(\mathrm{n})$.

2. Master Theorem

a. Algorithm A has a recurrence of the form $T(n) = 9T(\frac{n}{3}) + \theta(n^2)$. We consider the value $c = \log_b a = \log_3 9 = 2$. Since $f(n) = \theta(n^2) = \theta(n^c)$, we have the case of the master theorem where $T(n) = \theta(n^c \log n)$. Therefore, this has a time complexity of $\theta(n^2 \log n)$.

b. Algorithm B has a recurrence of the form T(n) = 2T(n-1) + O(1). This does not fit the Master Theorem, but we observe that the number of subproblems will double n times. Each subproblem will take constant time, so this is a time complexity of $\theta(2^n)$.

c. Algorithm C has a recurrence of the form $T(n) = 5T(\frac{n}{2}) + \theta(n)$. We consider the value $\log_b a = \log_2 5$. Since $f(n) = \theta(n^c)$ where $c = 1$, and $1 < \log_2 5$, we have the case of the master theorem where $T(n) = \theta(n^{\log_b a})$. Therefore, this has a time complexity of $\theta(n^{\log_2 5})$.

3. Local Minimum

Main Idea: Start at the root of the tree and probe its children. Return the root if it is the minimum, and if not, recursively call on a child.

Pseudocode:

```
LocalMin(T){
        If T has no children
                Return T
        Find T's value x(T) and its left and right children's values x(L), x(R)
        If x(T) < x(R) and x(T) < x(L)
                Return T
        If x(L) < x(T)
                Return LocalMin(L)
        Return LocalMin(R)
}
```

Proof of correctness:
First, we establish the property that T's parent will have a larger value than T. Take the base case, where T is the root. T is considered to have a smaller value than its parent by default. In the body of the function, we return if T is larger than both of its children. We recursively call LocalMin() on a node with value less than T (if both children had a value greater than T we would have returned earlier).

Since T always has a smaller value than its parent, we return T if it has a smaller value than both children. By definition, T in this case is a local minimum. Eventually, the recursion will end if we reach a leaf of the tree, or the case in which T has no children. Since we have established the property that T has a smaller value than its parent and T has no children, it is by definition a local minimum.

Complexity:
Probes: At each iteration of the function, there will be a maximum of 3 probes (T, T's left child, T's right child). Each step of recursion traverses one level of the binary tree, which is equivalent to the height. Therefore, there will be $O(3\log n) = O(\log n)$ probes.

4. Largest Overlap

Main Idea: First, sort by s_i. Then, use a divide and conquer approach, by splitting into two roughly equal halves. the largest overlap will either be in the left half, the right half, or exist in the overlap in the middle.

Pseudocode:

```
Pre-Sort intervals by s_i

LargestOverlap([s_1, f_1], ... , [s_n, f_n]){
        // Only one interval, so no overlap
        If i == 1
                Return 0
        Middle = ⌊ n/2 ⌋
        LeftInterval = [s_1, f_1], ... , [s_middle, f_middle]
        RightInterval = [s_middle+1, f_middle+1], ... , [s_n, f_n]
        LeftOverlap = LargestOverlap(LeftInterval)
        RightOverlap = LargestOverlap(RightInterval)
        // Merge
        LargestLeft = negative infinity
        For j from i to middle
                LargestLeft = max(LargestLeft, f_j)
        MergeOverlap = 0
        For k from middle+1 to n
                If  f_LargestLeft < s_k
                        Do nothing
                Else If f_LargestLeft <= f_k
                        MergeOverlap = max(MergeOverlap, f_LargestLeft - s_k)
                Else f_k < f_LargestLeft
                        MergeOverlap = max(MergeOverlap, f_k - s_k)

        Return max(LeftOverlap, RightOverlap, MergeOverlap)
}
```

Proof of correctness:
We use a divide-and-conquer approach for this problem. First, we pre-sort the intervals by s_i. The base case, with just one interval, returns 0 overlap, which is correct. Then, we subdivide the problem into two problems with half the input size, and recursively call the problem. The largest overlap is either the largest overlap found in the left half, right half, or in the overlap between the left and the right half by the property of divide-and-conquer.

We consider the merging part of this algorithm. We identify the largest f_i value of the left side. This places a hard-right bound on any interval found by merging, as any interval that exceeds this point cannot contain overlapping intervals from the left and right sides. We also notice that because we sorted by s_i, the starting value s_k for any k that we consider will be the limiting factor on the left side. We iteratively go through each interval in the right half. There are three cases that we now consider. If $f_{\text{LargestLeft}} < s_k$, then there is an overlap of 0. This is because the right side interval that we are comparing begins after the latest interval from the left side ends. Therefore, we do not update the overlap (which is initialized to 0). If $f_{\text{LargestLeft}} <= f_k$, then we know that the interval must end at $f_{\text{LargestLeft}}$. Because the intervals were pre-sorted by s_i values, we are guaranteed that element k will begin after all values in the left half. Therefore, in order to calculate the overlap, we must take the length of the interval $[s_k, f_{\text{LargestLeft}}]$. Finally, we are left with the case in which $f_k < f_{\text{LargestLeft}}$, which means the entire interval $[s_k, f_k]$ is contained within interval on the left side (because of the sorting of s_i). This search through the right side is collectively exhaustive.

Now consider the possibility that there is a longer interval that could have been generated by picking a different interval from the left side (other than LargestLeft). Because we searched through the entire left side to pick the latest end point, we know that this must allow for a larger than or equal to overlap than any other possible interval. Because we sorted the array by s_i, we know that right side interval will have a larger or equal starting point s_k than any starting point from a left side interval. Therefore, the overlap will be limited by s_k, and a longer interval by considering a different interval is not possible.

Complexity:
Runtime: We pre-sort the intervals, which requires O(nlogn) time using an efficient sorting algorithm (merge sort, heap sort, etc.). Then, we use a divide-and-conquer approach, dividing the problem into two halves that are to be solved recursively. The merging part of the algorithm requires linear time, as we iterate through each half of the intervals to find the LargestLeft value, and then the other half of the intervals to run comparisons. Therefore, we obtain the recurrence relation $T(n) = 2T(\frac{n}{2}) + \theta(n)$. By the Master Theorem, this is θ(nlogn). Combining this with the sorting gives us an overall time complexity of O(nlogn).
Space: The sorting algorithm has a space complexity of O(n). Each function call of the recursive algorithm will take $O(1)$ space because we are just allocating variables. The recursion will go to a maximum depth of logn, giving a space complexity of O(logn). Therefore, the overall space complexity is O(n).