Problem 1

Answer:

Algorithm:

The algorithm is divided into two functions.The first function calculates the minimum number of days(result) from the n festival to cause the rain on given maximum duration between two consecutive rains(mid).

```
findRain(int arr[],int mid, int n){
  int currentRain = arr[0];
  int result = 1;
  for(int  i = 1; i<n; i++){
      if(arr[i] - currentRain >= mid){
          currentRain = arr[i];
          result++;
      }
  }
  return result;
```

The second function uses the above function to search for the relevant index in the above calculated values using a binary search.This function returns the minimum duration between the rains that is maximized.

```
MinimumDistance(int arr[],int n, int k){
  int left = 0; int right = arr[n-1];
  int output = -1;
  while(left<right){
      int mid = (left+right)/2;
      int minTower = findRain(arr,mid,n);
      if(minTower >= k){
          output = Math.max(output,mid);
          left = mid+1;
      }
      else{
```

```
            right = mid;
        }
    }
    return output;
}
```

Proof of Correctness:
For the first function(findRain):We just rain on the very first day(Assumption given in the question) and then we proceed by brute force raining on any day where the difference between current Day and the last rain day >= maximum allowed.
Since this is brute force, this gives us the minimum number of days to rain on given a maximum duration between any two rains allowed.

Now assume an array made up by calculating values from the above function,the index being the maximum duration allowed between any two rain days.i.e. the very first element(index 0) will have the number of rain days such that maximum duration between any two rains is 0.
It can be clearly observed that any such array would be sorted in decreasing order since more duration allowed means less number of rain days required.
Once we've constructed such an array then:
Claim 1: Our answer will be the rightmost element in the above array such that it's value is greater than or equal to k.
Proof of Claim1:
Proof by Contradiction: Assume that our answer is not the rightmost element such that it's value is greater than or equal to k.It is some other element other than the above one(Let this element be X).
Let the rightmost element in the array such that it's value is greater than or equal to k be Y.
Case1:X is present on the right of Y
In this case, X is not even a feasible solution since it is raining on less than k days.
Case 2: X is present on the left of Y

In this case, we can always choose an element on the right side of X such that it's value is greater than or equal to k(There will always be such an element present in the right side of X since in this case X is present on the left side of Y).Now picking such an elements will increase the maximum duration between any two rain days since we are going on the right side of the array and the index is increasing and index is the maximum duration between any two rain days.This contradicts the optimality of X since there is another solution which is better than X.
Similarly we can keep going further right until we land on Y, which will be the final solution.
Hence claim 1 is proved by Contradiction.

We don't need to construct the whole array since we know that it's elements will be arranged in decreasing order by default, we can just apply binary search on the indexes and calculate the values on that step itself.

Time Complexity:
Since the above function(MinimumDistance), uses binary search,hence it makes O(log(n)) calls.And in these log(n) calls, we do work O(n)(Work done inside findRain function.)
So final time complexity: T(n) = O(nlog(n))

Problem 2

a.

Answer:

We can compute the maximum-weight spanning tree of a given edge-weighted graph using prim's algorithm by making just a minor modification in the algorithm. Instead of picking the cheapest edge from X and V-X, we pick the most expensive edge instead.

Algorithm:

S ← (source Vertex(randomly picked))
X ← {S}, F ← ϕ
while(X ≠ V){
    e = (u,v) ← (most expensive edge with u ∈ X and v ∈ V-X)
                (implemented using maxHeap)
    F = F ∪ {e}
    X = X ∪ {v}
}
return F

Justification:

This can be proved similarly as the proof of the cut lemma, just instead of contradicting for the cheapest edge in Minimum Spanning Tree, we can contradict for the most expensive edge in Maximum Spanning Tree.

The rest of the parts such as the tree being acyclic and connected can be proved similarly using the proof of prim's algorithm.

b.

Answer:

Proof by contradiction:

Suppose that the path between any two vertices u and v is not the widest paths between these vertices.Suppose there is some other path(Let's call it P)

Now,let u belong to X and v belong to V-X.Let any vertices that are in path P be randomly distributed in X and V-X.

Since this is not the widest path,and there must be one edge connecting these two sets (X and V-X) together, we can add another edge connecting these two sets from the widest path between u and v.Since a spanning tree is connected, adding a new edge will create a cycle.Now let us remove the edge from Path P that was connecting X and V-X.Since, path P is not the widest path,our new edge must have cost strictly greater than the the edge that we removed.Doing this increased the cost of the path between u and v in the maximum spanning tree which is a contradiction of how a maximum spanning tree is defined.

Hence, the path between any two pair of vertices is the widest path in a maximum spanning tree.