ASSIGNMENT 3:

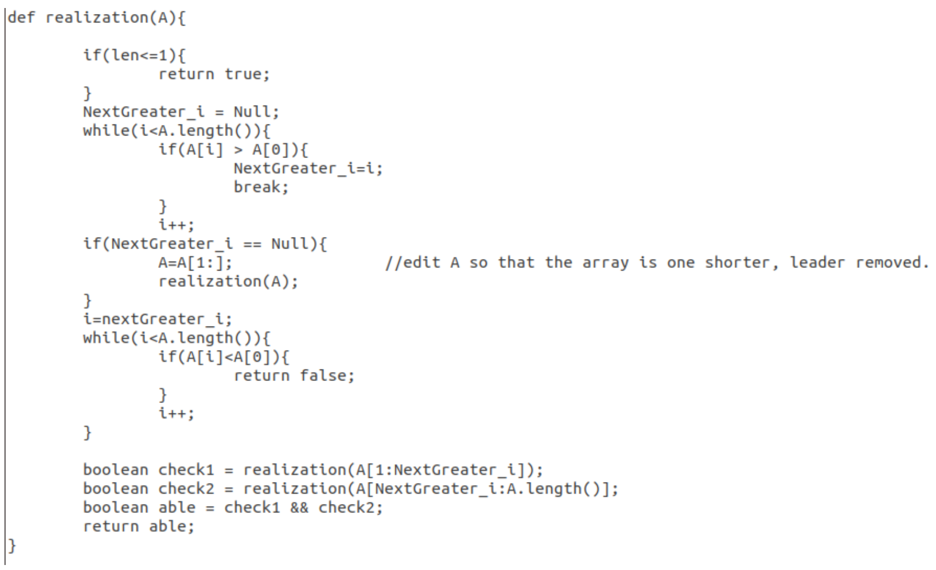
In the task we were asked to Design an algorithm for testing if there exists a pre-order realization T of a given array A such that the in-order traversal of T visits the values of A in sorted order.

When thinking about the arrays and looking at the structure of the examples given, it is important to note that for an in-order traversal to work, then this implies having a binary tree. Seeing the tree would have to be initially constructed using pre order traversal this implied that the first element in the array must be the root of the tree.

After working out and observation I believe I have found an algorithm that can successfully detect whether, a realization can be created or not and this came from the observation of in the examples that the order would have something to do with splitting the numbers so that the next greatest number after the root would be the start of the next branch. This would happen on the various subtrees throughout the project.

In short, the pattern found and explored in the pseudocode below is based off the following pattern: Building the tree using the provided array and once it breaks a condition, return false. The pattern is that, for any subtree in the tree, and given the root of the subtree, all values of the root must be smaller than the root, and all values on the right must be larger.

Once finding this pattern I was able to use this to create the following pseudocode:



```
def realization(A){

        if(len<=1){
                return true;
        }
        NextGreater_i = Null;
        while(i<A.length()){
                if(A[i] > A[0]){
                        NextGreater_i=i;
                        break;
                }
                i++;
        if(NextGreater_i == Null){
                A=A[1:];                //edit A so that the array is one shorter, leader removed.
                realization(A);
        }
        i=nextGreater_i;
        while(i<A.length()){
                if(A[i]<A[0]){
                        return false;
                }
                i++;
        }

        boolean check1 = realization(A[1:NextGreater_i]);
        boolean check2 = realization(A[NextGreater_i:A.length()];
        boolean able = check1 && check2;
        return able;
}
```

This code essentially utilises the observations stated above, checking each subtree whether the pattern identified remains. We can see this when using the example array in the assignment brief, <2,1,4,3>.

The first thing that happens is that it bypasses the first if statement because it has length 4

Then it will get the next greatest number after the first element 2, identify that it is 4 and take its index to be 2 (as it’s the third in the list)

It will then bypass the statement checking that the first element isn't the greatest in the list.

The next code will then check in the range from index, 2 to 4 which is the nextGreatest index to the end of the list and see if any number in there is smaller than the root (the original first element) if this is the case then creating this realisation is not possible and will consequently return false.

The final piece of code is the reiterative step, which will run in linear time, to split the original array up into two lists by the next Greatest index as defined earlier. This step will run the whole function from the beginning to repeat the process to also consider the subtrees. If it gets to the end, and there hasn't been a false throughout the iterations then the Booleans will come together to return true, indicating that the realization is possible from the original array.

PROVING CORRECTNESS:

In order to prove this algorithms correctness then we could consider that the way in which I have been thinking about the problem could potentially be wrong, so assuming that my original premise was incorrect, being that in any subtree of a given tree, the left child must be less than the parent and the right child must always be more than the parent. If we were to negate this and say the opposite is true, that it is not the case that the left child is smaller than the parent OR it is not the case that the right child is more than the parent.

In order to disprove this new premise and thus prove the original algorithms correctness then I just need to show that a left child cannot be more or equal to in value to its parent.

When looking at the example shown before with the algorithm A=<2,1,3,4>. We must assume that in this example that there can be a realization in which the left child is more than the parent. When creating the tree in a pre order traversal we can see that this is actually impossible which must mean that there is no possible way that there could be a binary tree that the in-order could possibly take in which there would be a node with a left child of a greater value than the parent, thus disproving this new assumption and proving that the original concept that the algorithm was based off to be correct.

TIME COMPLEXITY:

As it clearly states in the lecture slides, that recursive functions are always a factor of linear time. Although this means that the algorithm could recurse over and over again, this would mean that it would only be a factor of linear time complexity. All the loops used in the algorithm were not nested meaning that they would maintain a time of O(n). To analyse the time complexity here, we must consider the worst case. At worst case time for this algorithm the most it will have to execute will be 2 loops running at the same time, as these are both complexity of O(n) then the worst case scenario would mean this program running in O(n^2).