Theoretical Assignment - 2

ESO207

Name: Aaryaman Aggarwal

Roll No.: 230020

September 2025

1. (a). It is evident from the recursion tree that $k \leq 2 * n - 1$ as the maximum number of elements in the recursion tree can be $2n-1$ if the array is completely reversed, that is: $[n, n-1, n-2, \ldots, 2, 1]$.
Furthermore, since it is a complete binary tree due to the fact that each call of *ArchiveSort* can only incur 0 or 2 more calls of the function.
It can also be observed that if we swap the two middle elements of the sorted array or the sub-array, the function will be called 2 more times. The following algorithm can be proposed. [1em]

Algorithm 1 Permutations

```
Maintain a global variable CountSwaps initialized to 0
function PERM(Array,l, r, n)
    if CountSwaps = (k-1)/2 or n = 1 then
        Return;
    end if
    mid ← (l+r)/2
    Swap(A[mid], A[mid + 1])
    CountSwaps ← CountSwaps + 1
    PERM(arr A,l, mid, n/2)
    PERM(arr A, mid+1, r, n − n/2)
end function
function CALCULATE
    if k mod 2 = 0 OR k > 2 · (r − l + 1) − 1 then
        return No
    end if
    Create an array A[0, 1 · · · n − 1]
    A[i] ← i + 1 ∀i = 0, 1, ...n − 1
    PERM(A, 0, n-1, (k-1)/2)
end function
```

(b). Our assertion P(i) is that after i calls of the swaps of mid and mid+1 terms of the

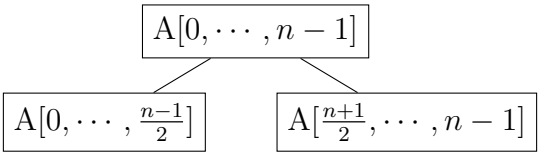
array and subsequent sub-arrays we will have 2i+1 calls of the function.

$$P(i) : \text{FunctionCalls} = 2i + 1$$
$$i = \text{number of swaps}$$

Proof by method of induction,

Base case:

P(1): If we do 1 swap of the middle terms of the entire (n-1)/2 and (n+1)/2 term. Archive sort is called for $\text{A}[0, \cdots n-1]$, which leads to $\text{A}[0, \cdots (n-1)/2]$ and $\text{A}[(n+1)/2, \cdots n-1]$. The function stops here since this split of array is already sorted. So, we do 3 calls of the function.



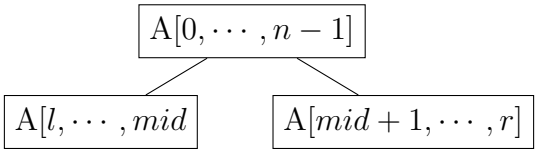
Induction hypothesis:

Assume that this hypothesis is true for all $i < k$.

Inductive step:

Assuming that the hypothesis is true for all $i \leq \frac{k-1}{2}$ If i swaps have already been done, it means that Archive sort will be called $2i+1$. If we swap the middle two elements of any sub-array available after i swaps, then the function will be called two more times. Consider the arbitrary sub-array $\text{A}[l, \ldots, r]$ which is already sorted and hence will not invoke any further calls of archive sort.

If we swap the middle elements of this array, it will lead to two more function calls as follows:



Hence, it is evident that two more function calls are invoked as another swap is made and our induction holds true.

$$P(i) : \text{FunctionCalls} = 2i + 1$$
$$i = \text{number of swaps}$$
$$i \leftarrow i + i$$
$$P(i+1) : \text{FunctionCalls after an additional swap} = 2i + 3$$
$$\implies 2(i+1) + 1$$

(c). Time complexity Analysis:

Assume the time complexity is T(n) where n is number of swaps.

$$T(n) = 2T(n/2) + c$$

By unfolding we get,

$$T(n) = c + 2c + 4T(n/4)$$

$$T(n) = c\frac{2^{logn} - 1}{2 - 1}$$

$$T(n) = c(n - 1)$$

$$T(n) = \mathcal{O}(n)$$

Another time-complexity analysis is that since the function $Perm$ can only be called a maximum number of $\frac{k-1}{2}$ times and the steps in the function are of constant time using the following we can say that:

$$k \leq 2n - 1$$

$$\implies \frac{k-1}{2} \leq n - 1$$

$$T(n) \leq c(\frac{k-1}{2})$$

$$\implies T(n) \leq c(n - 1)$$

$$\implies T(n) = \mathcal{O}(n)$$

2. The idea is that at every node we calculate what is the strength value if we include the node and if we exclude it. If we choose to exclude the current node the strength of this choice is the maximum of what we can get from the sub-tree. The function returns the strength of the subtree in two-cases:

- If the said node is included, and the two children are not included(Line 7)
- If the said node is not included, then the maximum of including the left child, excluding the left child, added to the maximum of including the right child and excluding the right child (Line 8).

The proposed algorithm is:

(a).

Algorithm 2 Royal Guard Deployment

```
1: function MaxStrength(node)
2:     if node is NULL then
3:         return (0, 0)
4:     end if
5:     (LeftInc,LeftExc) ← MaxStrength(node.left)
6:     (RightInc,RightExc) ← MaxStrength(node.right)
7:     include ← node.strength + LeftExc + RightExc //Both children must be excluded
8:     exclude ← max(LeftInc,LeftExc) + max(RightInc, RightExc)
9:     return (include,exclude)
10: end function
11:
12: function Strength(root)
13:     (incRoot, excRoot ) ← MaxStrength(root)
14:     return max(incRoot,excRoot)
15: end function
```

(b). At every function call of the we compute the include and exclude for only one node. The overall algorithm only visits each node once. Hence, each node is basically traversed once and the time complexity is of O(n).

3.

(a). The brute force approach is to fix every elements maximum as the peak, and then continue filling in the left and right of it on the following basis:

$$Peak = a_p = m_p$$
$$\text{for } i < p \implies a_i = min\{m_i, a_{i+1}\} \text{for } i > p \implies a_i = min\{m_i, a_{i-1}\}$$

This essentially checks every possible permutation of building floors possible.
[1em] The approach is that we find all the possible ways the buildings can be built while

Algorithm 3 Pretty buildings - Brute force

```
1: function BestSum(m[0, 1, ..., n − 1], n)
2:     BestSum ← 0
3:     Initialize array FinHeights[0, 1, ..., n − 1] to store final heights
4:     Initialize array a[0, 1, ..., n − 1] to store heights during computation
5:     for p ← 0 to n − 1 do
6:         sum ← m[p]
7:         a[p] ← m[p]
8:         for i ← p − 1 to 0 do
9:             a[i] ← min(a[i + 1], m[i])
10:            sum ← sum + a[i]
11:        end for
12:        for i ← p + 1 to n − 1 do
13:            a[i] ← min(a[i − 1], m[i])
14:            sum ← sum + a[i]
15:        end for
16:        if sum > BestSum then
17:            BestSum ← sum
18:            Copy array a to the array FinHeights
19:        end if
20:    end for
21:    Return FinHeights, BestSum
22: end function
```

keeping in mind the conditions given.

Time complexity analysis:
In the given algorithm, we have one loop that iterates n-times and within that loop we have two loops that iterate $(n-1)$ times cumulatively. Hence the time-complexity would be:

$$T(n) = c_1 * n * (n-1) + c_2$$
$$T(n) = \mathcal{O}(n^2)$$

(b). For the most optimal solution without dips we have a sequence which is either non-decreasing for the entire array or non decreasing upto a index and then non increasing continuously until the end of the array. So, we try to find this peak index. We find this index by computing the maximum sum obtained from the left hand side and the maximum sum obtained from the right hand side upto each index using 2 stacks and then find the index for which this sum is maximum and then we choose the maximum possible height at each index for the others based on fixing the found index to its maximum possible height already given. Once we've found the peak for our buildings, we fill in the rest of the array using the same logic as before. We define the following two arrays $left[0, 1, ..n-1$ and $right[0, 1, ..., n-1]$:

$$left[i] = \sum_{j=1}^{i} min(m_j, m_j + 1, ..., m_i)$$

$$right[i] = \sum_{j=i}^{n} min(m_i, m_i + 1, ..., m_j)$$

$$score(i) = left[i] + right[i] - m_i$$

Whichever value of i maximizes $score(i)$ is the location of the peak. Hence, we use the stack to compute the running minimum to calculate the above sum.

Algorithm 4 Pretty buildings using Stack

```
1:  function COMPUTE(m[0, 1 ··· n],n)
2:      Create a stack named stack which holds pairs (height, count)
3:      Create two arrays left[0, 1 ··· n − 1] and right[0, 1 ··· n − 1]
4:      //For Left Pass
5:      sum ← 0
6:      for i ← 0 to n-1 do
7:          count ← 1
8:          while stack.empty() is Not True and stack.top().height > a[i] do
9:              sum ← sum - stack.top().height * stack.top().count
10:             count ← count + stack.top().count
11:             stack.pop()
12:         end while
13:         sum ← sum + m[i] * count
14:         stack.push(m[i], count)
15:         left[i] ← sum
16:     end for
17:     clear stack
```

```
    //For right pass
    sum ← 0
    for i ← n-1 to 0 do
        count ← 1
        while stack.empty() is Not True and stack.top().height > a[i] do
            sum ← sum - stack.top().height * stack.top().count
            count ← count + stack.top().count
            stack.pop()
        end while
        sum ← sum + m[i] * count
        stack.push(m[i], count)
        right[i] ← sum
    end for
    // To calculate the max
    max ← 0, k ← 0
    for i ← 0 to n-1 do
        if left[i]+right[i]-m[i] > max then
            max ← left[i]+right[i]-m[i]
            k ← i
        end if
    end for
    for i ← k-1 to 0 do
        a[i] = min(m[i],a[i+1])
    end for
    for i ← k+1 to n-1 do
        a[i] = min(m[i],a[i-1])
    end for
end function
```

Time complexity analysis:

As we can see in the first two for-loops where the arrays $left[n]$ and $right[n]$ are calculated, in each pass, every element is popped or pushed at most once. Hence, the first two loops can have at max $2n$ number of instructions. Therefore, each of these two loops can only have a max of $c_1 n$ instructions.

The loop to calculate the maximum $score(i)$ also iterates only n times ith a constant number of instructions inside the loop. Hence, this can also only incur $c_2 n$ number of instructions.

Lastly, the loop to finally construct the arrangement of towers also iterates $(n-1)$ times cumulatively with a constant number of instructions $c_3(n-1)$. Therefore the

time-complexity is as follows:

$$T(n) = c_1 n + c_2 n + c_3(n-1)$$
$$\implies T(n) = \mathcal{O}(n)$$