ESO 207 THEORETICAL ASSIGNMENT 2

CHANDRADIP KARMAKAR
240296

NAME: CHANDRADIP KARMAKAR ROLL NO: 240296

Question 1: The Archive Sorting Ritual

(a) Algorithm:

Here we can assume Each call to ArchiveSort is a node in a recursion tree where a node either (A) is a leaf (the segment is already strictly increasing), or (B) has exactly two children (we split and recurse).

We build array `ans[pos .. pos+n-1]` containing the integers `startVal .. startVal+n-1`.

If `k==1` we must make the whole segment already strictly increasing fill `ans[pos+i] = startVal+i` for `i = 0...n-1`.

If `k>1` the root call must rescue: total calls = `1+ k_left + k_right` where `k_left` is the number of calls consumed inside left half and `k_right` inside right half. Let:

$$\text{left\_size} = \lfloor (n+1)/2 \rfloor$$
$$\text{right\_size} = n - \text{left\_size}$$

AND feasible `k_left` $\in [1, 2 \times \text{left\_size} - 1]$ (odd), `k_right` $\in [1, 2 \times \text{right\_size} - 1]$ (odd), and `k_left + k_right = k - 1`.

Now we have to choose such a split (always possible when `k` is feasible). To ensure the parent segment is not strictly increasing, we place large numbers in the left block and smaller numbers in the right block. So basically:

- give the left child the `left_size` largest numbers from the current range, i.e. `startVal + right_size .. startVal + n - 1`.
- give the right child the `right_size` smallest numbers `startVal .. startVal + right_size - 1`.

And this guarantees that some left element > some right element so the whole segment is not strictly increasing.

Thus recursively build the left and right parts with the chosen `k_left` and `k_right`.

Pseudocode:

```
Pos = 0;
Array ans;

build(pos, n, k, startVal) {
    if (n == 0) return;
    if (k == 1) {
        // Make the segment strictly increasing: startVal, startVal+1,
 ...
        for (int i = 0; i < n; i++) ans[pos + i] = startVal + i;
        return;
    }

    int left_size = (n + 1) / 2;
    int right_size = n - left_size;
    int x = k - 1; // amount to distribute to children (k_left +
 k_right)
```

```

    // child k ranges: k_left in [1, 2*left_size -1], k_right in [1, 2*right_size -1]
    int low = max(1, x - (2*right_size - 1));
    int high = min(2*left_size - 1, x - 1);

    if (low > high) {
        low = 1;
        high = min(2*left_size - 1, x - 1);
    }

    int k_left = high;
    if (k_left % 2 == 0) k_left--;
    if (k_left < low) {
        k_left = low;
        if (k_left % 2 == 0) k_left++;
    }

    int k_right = x - k_left;

    // Assign numbers: left gets the largest left_size numbers of the current range
    int leftStartVal = startVal + right_size;
    int rightStartVal = startVal;

    // Fill left block (positions pos .. pos+left_size-1)
    build(pos, left_size, k_left, leftStartVal);

    // Fill right block (positions pos+left_size .. pos+n-1)
    build(pos + left_size, right_size, k_right, rightStartVal);
}
```

Thus the `ans` array stores the required permutation.

(b) Proof of correctness:

The recursion exactly mirrors ArchiveSort decision structure: `k==1` corresponds to making the segment strictly increasing (so that node is a leaf). `k>1` forces that node to split and consume `1 + k_left + k_right` calls. The choice of distributing `k-1` among children is always possible because of the ranges described earlier. As we brand all larger numbers in the left half and smaller in the right, the combined segment is guaranteed **not** strictly increasing, so ArchiveSort will make the two recursive calls at that node.

So now by Induction on `n` we can prove that the produced permutation indeed causes exactly `k` calls.

(c) Time complexity:

In the `build` function: checking `k==1`, computing split sizes, picking `k_left`, `k_right`. So for that the time complexity is $\mathcal{O}(1)$.

If `k==1`, it fills the segment with consecutive values $\to \mathcal{O}(n)$ for that leaf segment, since it writes `n` numbers into `ans`.

If `k>1`, it makes two recursive calls, splitting the range into left and right halves (sizes roughly `n/2` and `n/2`).

Now each element of the output permutation is written exactly once:

1. Either in a leaf(`k==1`)
2. Or deeper down the recursion tree, but still exactly once.

Thus the overall time complexity of the above discussed algorithm is $\mathcal{O}(n)$.

Question 2: Royal guard deployment:

(a) Algorithm and pseudocode:

Here, first of all we have given a binary search where each node having key and strength value. So here we have to find a subset of key so that the sum of corresponding strength value will be maximum. But there is a constraint that we cannot select both parent and its child in our output subset. So, basically we have to find the subset of non-adjacent nodes of the BST having the maximum sum of strength values.

Here we can use TAKE or LEAVE approach where in case of TAKE, we first take the current root. As we take it, we cannot take its children as they are adjacent nodes, so we have to take the next level of its child node. And in case of LEAVE, we leave the current node. As we leave the current node, we can take the children nodes. Thus we can make recursive calls to find the the maximum value that can come from the binary tree rooted at a particular node and accordingly the subset according to which we have to choose to get the maximum strength value sum.

But here is a problem. As we are making recursive calls, we call same functions multiple times to get the same maximum value coming from a particular node. This increases the time complexity from $\mathcal{O}(n)$ to exponential time complexity of the approach.

This to avoid the overlapping subproblems, we can store the maximum value coming from a particular node to some data structure like map or Red Black Tree, and can check that when the function gets the root that already exists in the data structure we can directly return the value rather than solving the function again.

Pseudocode:

```
function solve(root):
    if root is null:
        return 0
    if root is in memory:
        return memory[root]

    // ll, lr, rl, rr represent the results from grandchildren
    int ll = (root->left && root->left->left) ? solve(root->left->left) : 0;
    int lr = (root->left && root->left->right) ? solve(root->left->right) : 0;
    int rl = (root->right && root->right->left) ? solve(root->right->left) : 0;
    int rr = (root->right && root->right->right) ? solve(root->right->right) : 0;

    int take = root->data + ll + lr + rl + rr;
    int leave = solve(root->left) + solve(root->right);

    int z = max(take, leave);
    memory[root] = z;
    return z;
```

Thus the function gives the possible maximum sum of the strength values. Also we can store the subset by storing the node which is contributing in the maximum sum in the recursive function.

(b) Time complexity:

Without storing the maximum value coming from a particular node, we are doing overlapping subproblems in the recursive calls. Thus it may take exponential time complexity. But with storing the maximum values in the data structure, we are iterating each node only once. When in the recursive calls, we are getting a node that is already present in the data structure, we simply return the value from the data structure without solving the function again. Thus the overall time complexity of the algorithm is $\mathcal{O}(n)$, where `n` is the number of nodes present in the BST.

Question 3: As pretty as it gets

(a) Bruteforce approach:

For bruteforce approach, we have to check all possible combinations of the heights. So, for each hall `i`, its height `a_i` can range from `1` to `m_i`.

Also after making the combinations of the heights, we have to check whether there is a dip present in the array or not by iterating through all the elements of the array. If `a_j > a_i` and `a_k > a_i` for some `j < i < k`, then the sequence is invalid.

After validating the sequence, we have to calculate the sum of the elements of the sequence. If the sum is greater than the current maximum sum found so far, update the maximum sum and store the sequence.

Time complexity:

First we have to generate all possible combinations. For each element, it has `m_i` choices, so in the worst case, if `M=max(m_i)`, there will be roughly $\mathcal{O}(M^n)$ possible sequences.

For each sequence, checking the no dip condition involves three nested loops: `i` (from `2` to `n-1`), `j` (from `1` to `i-1`), and `k` (from `i+1` to `n`). This takes approximately $\mathcal{O}(n^3)$ time for each sequence.

For taking the sum, it will take $\mathcal{O}(n)$ time. So, overall time complexity of the bruteforce approach is $\mathcal{O}(M^n \cdot n^3)$.

(b) Efficient algorithm:

As there should not be any dip in the array of heights, so there should be one peak in the array of heights. Meaning, the left part of the peak is increasing and the right part of the peak is decreasing. Now as the sum should be maximum, so from the peaks we have to find the optimal peak, that gives the maximum sum of the heights.

For that, we have to precompute the `leftsum` and the `rightsum` in such a way that the left part of peak at each index is increasing and the right part of the peak at each index is decreasing. And we can do this precomputation efficiently using stack in linear time.

Pseudocode for `leftsum`:

```
function computeLeftSum(m[1..n]):
    stack <- empty
    LeftSum[0] <- 0 // Assuming 1-indexed array for m

    for i = 1 to n:
        while stack not empty AND m[stack.top] >= m[i]:
            stack.pop()

        prev = 0
        if stack not empty:
            prev = stack.top

        LeftSum[i] = m[i] * (i - prev) + LeftSum[prev]
        stack.push(i)

    return LeftSum
```

Pseudocode for the `rightsum`:

```
function computeRightSum(m[1..n]):
    // Reverse the array m to compute right sum as left sum
    mRev = new Array[n+1]
    for i = 1 to n:
        mRev[i] = m[n - i + 1]

    LeftSumRev = computeLeftSum(mRev)

    RightSum = new Array[n+1]
    for i = 1 to n:
        j = n - i + 1
        RightSum[i] = LeftSumRev[j]

    return RightSum
```

Now we have to find the best peak from all the indices using the precomputed `leftsum` and `rightsum` array.

Pseudocode for finding the best peak:

```
function findBestPeak(m[1..n]):
    LeftSum = computeLeftSum(m)
    RightSum = computeRightSum(m)

    bestSum = -infinity // Initialize with a very small number
    bestPeak = 1 // Initialize with an arbitrary peak index

    for p = 1 to n:
        S = LeftSum[p] + RightSum[p] - m[p] // Subtract m[p] because it
    's counted twice
        if S > bestSum:
            bestSum = S
            bestPeak = p

    return bestPeak
```

Now using the best peak, we can now reconstruct the required optimal heights to get the final output array.

Pseudocode for reconstruction:

```
function reconstructHeights(m[1..n], bestPeak):
    a = new Array[n+1]
    a[bestPeak] = m[bestPeak]

    // Fill to the left
    for i = bestPeak - 1 down to 1:
        a[i] = min(m[i], a[i+1])

    // Fill to the right
    for i = bestPeak + 1 to n:
        a[i] = min(m[i], a[i-1])

    return a
```

Thus, we can find the output array `a`.

(c) Time complexity analysis of the optimized algorithm described above:

Here, we are using stack for precomputation of the `leftsum` array and the `rightsum` array. So we check for each index whether left part is increasing and the right part is decreasing and accordingly we push or pop elements in the stack only once. Hence the time complexity of getting the `leftsum` and the `rightsum` array is $\mathcal{O}(n)$, where `n` is length of the given array as each index is pushed or popped once. Also for finding the best peak from the `leftsum` and `rightsum`, we have to iterate through all the elements of the arrays. So hence the time complexity for it is $\mathcal{O}(n)$. Again for reconstructing the array, we have to go to left and right from the `bestPeak` index. So here also the time complexity will be $\mathcal{O}(n)$.

Hence, the overall time complexity of the algorithm is $\mathcal{O}(n)$.