ESO207 – Theoretical Assignment #2

Name: Sumath Rengasami V

Roll No: 241054

Date: September 26, 2025

Question 1. The Archive Sorting Ritual

(a) Algorithm and Construction

Feasibility. Let `calls`$(A[l..r])$ be the number of invocations made when `ArchiveSort` is run on $A[l..r]$.

- If $A[l..r]$ is strictly increasing, then exactly 1 call is made.
- Otherwise, we split:

$$\texttt{calls}(A[l..r]) = 1 + \texttt{calls}(A[l..m]) + \texttt{calls}(A[m+1..r]).$$

Thus every expansion increases the number of calls by 2. Hence the number of calls is always odd, and the maximum possible is $2n - 1$ (when recursion goes to singletons).

$$\text{A solution exists iff } 1 \le k \le 2n - 1 \text{ and } k \text{ odd.}$$

Idea. We force recursion exactly where we want:

- At nodes where we want to **stop**, make the subarray strictly increasing.
- At nodes where we want to **expand**, make the subarray not strictly increasing by placing *larger values before smaller ones.*

Pseudocode.

```
ArchivePermutation(n, k):
    if k < 1 or k > 2*n - 1 or k % 2 == 0:
        return "IMPOSSIBLE"
    return Build(n, k, baseVal = 1)

Build(len, k, baseVal):
    if k == 1:
        // strictly increasing
        return [baseVal, baseVal+1, ..., baseVal+len-1]

    leftLen  = floor(len/2)
    rightLen = len - leftLen

    maxL = 2*leftLen - 1
    maxR = 2*rightLen - 1

    (kL, kR) = AllocateOdd(leftLen, rightLen, k)

    // Assign larger values to left to force "not increasing"
    leftBlock  = Build(leftLen,  kL, baseVal + rightLen)
    rightBlock = Build(rightLen, kR, baseVal)

    return Concatenate(leftBlock, rightBlock)
```

```
AllocateOdd(leftLen, rightLen, k):
    target = k - 1
    kL = 1
    kR = target - 1
    while kR > (2*rightLen - 1):
        kL += 2
        kR -= 2
    while kL > (2*leftLen - 1):
        kL -= 2
        kR += 2
    return (kL, kR)
```

—

(b) Correctness

We prove by induction on n that Build($n, k,$ baseVal) returns a block of consecutive integers arranged such that ArchiveSort makes exactly k calls.

Base case. If $k = 1$, the algorithm returns the numbers in increasing order. ArchiveSort recognizes this and stops immediately, making exactly one call.

Inductive step. If $k > 1$, we split into two halves and allocate odd k_L, k_R with $k_L+k_R = k-1$. By induction, each recursive call produces exactly k_L and k_R calls, respectively.

Since the left block contains strictly larger values than the right block, the concatenation is not strictly increasing, so ArchiveSort must expand at this node. Hence the total calls are

$$1 + k_L + k_R = 1 + (k - 1) = k.$$

The function AllocateOdd guarantees existence of such a split whenever $1 \le k \le 2n - 1$ and k is odd, since the maxima $2\ell - 1$ and $2r - 1$ of the subproblems cover the feasible range. Thus correctness is proved.

—

(c) Time Complexity

Case $k = 1$. The algorithm immediately returns the array in strictly increasing order. Constructing this output requires writing down all n elements once, which costs

$$O(n).$$

General k. Each element is placed exactly once into the permutation. The recursion tree has at most $O(n)$ nodes (since each node consumes at least one unit of k, and $k \le 2n - 1$). Each step does $O(1)$ extra work.

$$\text{Time complexity: } O(n), \qquad \text{Output space: } O(n).$$

—

Question 2. Royal Guard Deployment (30 points)

We are given a BST where each node has a *key* and a *strength* $w(u)$. We must select a subset of nodes (commanders) to maximize total strength subject to the Parent-Child Rule : if a node is selected, none of its children may be selected.

(a) Algorithm (in $O(n)$ time)

Idea. For each node u, compute two values:

$$\mathsf{in}[u] = w(u) + \sum_{v \in \text{children}(u)} \mathsf{out}[v], \qquad \mathsf{out}[u] = \sum_{v \in \text{children}(u)} \max\{\mathsf{in}[v], \mathsf{out}[v]\}.$$

in[u] is the best total if u is included (so children must be excluded). out[u] is the best total if u is excluded (so each child may be included or not, whichever is better). A single postorder traversal computes these in linear time. A second traversal reconstructs the chosen set.

Algorithm 1: RoyalGuard(root)

```
Input: Root of BST; each node $u$ has strength $w(u)$ and pointers $u$.left, $u$.right
       and $key = key(u)$
Output: Set $S$ of selected commanders maximizing total strength
Initialize in ←Map where each key is assigned 0; out ←Map where each key is
  assigned 0;
Function DFS($u, in, out$):
    if $u = null$ then
        return
    DFS($u.left, in, out$);    DFS($u.right, in, out$);
    if $u.left \neq null$ then
        $L \leftarrow key(u.left)$;
        $in[u.key] \leftarrow in[u.key] + out[L]$;
        $out[u.key] \leftarrow out[u.key] + \max(in[L], out[L])$;
    if $u.right \neq null$ then
        $R \leftarrow key(u.right)$;
        $in[u.key] \leftarrow in[u.key] + out[R]$;
        $out[u.key] \leftarrow out[u.key] + \max(in[R], out[R])$;
    $in[u.key] \leftarrow in[u.key] + w(u)$;
    return
DFS(root);
Function Reconstruct(u, takeParent, in,out):
    if $u = null$ then
        return
    if not takeParent and in[$u.key$] $\geq$ out[$u.key$] then
        add $u$ to $S$;
        Reconstruct($u$.left, true, in,out);
        Reconstruct($u$.right, true, in,out);
    else
        Reconstruct($u$.left, false, in,out);
        Reconstruct($u$.right, false, in,out);
$S \leftarrow \emptyset$;
if in[$root.key$] $\geq$ out[$root.key$] then
    add root to $S$; Reconstruct(root.left, true, in,out); Reconstruct(root.right,
     true, in,out);
else
    Reconstruct(root.left, false, in,out); Reconstruct(root.right, false, in,out);
return $S$;
```

(b) Time Complexity Analysis

Each node is processed a constant number of times:

- **DFS:** Each node does $O(1)$ work combining its (up to) two children $\Rightarrow O(n)$ time.
- **Reconstruct:** visits each node at most once with $O(1)$ work $\Rightarrow O(n)$ time.

Therefore, the total time is

$$O(n)$$

Question 3: As Pretty As It Gets!

We are tasked with assigning hall heights $a_1, a_2, \ldots, a_n$ for n plots, each bounded by

$$1 \leq a_i \leq m_i,$$

such that no hall looks like a "dip" between two taller halls. Formally, there must not exist indices $j < i < k$ with $a_j > a_i$ and $a_k > a_i$. The goal is to maximize the total height:

$$\max \sum_{i=1}^{n} a_i.$$

(a) Brute-force Algorithm

Idea. The "no dips" condition enforces that the skyline must be non-decreasing up to a peak, then non-increasing. For each possible peak position p, we can construct the tallest valid skyline and compute its sum.

Algorithm 2: Brute-force Skyline Construction

```
Input: Array of maximum heights $m[1..n]$
Output: Optimal hall heights $a[1..n]$ maximizing $\sum a_i$
bestSum ← $-\infty$;    bestConfig ← $\emptyset$ ;
for $p = 1$ to $n$ do
    initialize array $a[1..n]$;
    $a[p] \leftarrow m[p]$;
    for $i = p - 1$ 1 do
        $a[i] \leftarrow \min(m[i], a[i+1])$;
    for $i = p + 1$ to $n$ do
        $a[i] \leftarrow \min(m[i], a[i-1])$;
    currentSum ← $\sum_{i=1}^{n} a[i]$;
    if currentSum > bestSum then
        bestSum ← currentSum;
        bestConfig ← $a$;
return bestConfig;
```

Correctness. Suppose we fix a candidate peak p. The algorithm then builds the skyline outward from p, extending both left and right while always respecting the feasibility rules. At every step, it picks the largest height possible that still preserves unimodality: heights cannot decrease when moving toward the peak from the left, and they cannot increase when moving away from the peak on the right.

This design guarantees that no taller skyline can exist without breaking unimodality or violating feasibility. Since each position is set to the maximum feasible height, the total sum is also as large as possible for this choice of peak. Any other arrangement would necessarily make some column shorter, which lowers the overall sum.

Finally, because the global optimum must correspond to some peak position, trying all possible peaks and selecting the best skyline ensures we achieve the overall maximum. Thus, the algorithm is correct.

Complexity. Each peak construction requires $O(n)$ time for construction and $O(n)$ time for calculating sum. Trying all n peaks gives overall time complexity

$$O(2 \cdot n^2) = O(n^2).$$

(b) Efficient $O(n)$ Algorithm

For a given peak p, the optimal skyline sum is

$$S_p = \sum_{i=1}^{p} \min_{t=i..p} m_t \; + \; \sum_{i=p}^{n} \min_{t=p..i} m_t \; - \; m_p.$$

Define

$$L[p] = \sum_{i=1}^{p} \min_{t=i..p} m_t, \qquad R[p] = \sum_{i=p}^{n} \min_{t=p..i} m_t,$$

so that

$$S_p = L[p] + R[p] - m_p.$$

Algorithm 3: Left-to-right cumulative mins for $L[i]$

```
Input: $m[1..n]$
Output: $L[1..n]$
stack ← $\emptyset$;    tot ← 0;
for $i = 1$ to $n$ do
    $x \leftarrow m[i]$;    cnt ← 1;
    while $stack \neq \emptyset$ and $top.value \geq x$ do
        $(v, c)$ ← stack.pop();
        tot ← tot $-v \cdot c$;
        cnt ← cnt $+c$;
    push(stack, $(x, cnt)$);
    tot ← tot $+x \cdot cnt$;
    $L[i]$ ← tot;
```

Algorithm 4: Right-to-left cumulative mins for $R[i]$

```
Input: $m[1..n]$
Output: $R[1..n]$
stack ← $\emptyset$;    tot ← 0;
for $i = n$ to 1 do
    $x \leftarrow m[i]$;    cnt ← 1;
    while $stack \neq \emptyset$ and $top.value \geq x$ do
        $(v, c)$ ← pop(stack);
        tot ← tot $-v \cdot c$;
        cnt ← cnt $+c$;
    push(stack, $(x, cnt)$);
    tot ← tot $+x \cdot cnt$;
    $R[i]$ ← tot;
```

Algorithm 5: Find Optimal Heights in $O(n)$

```
Input: $m[1..n]$
Output: optimal $a[1..n]$
compute $L[1..n]$ by the left-to-right pass;
compute $R[1..n]$ by the right-to-left pass;
bestPeak $\leftarrow 1$;    bestSum $\leftarrow L[1] + R[1] - m[1]$;
for $p = 2$ to $n$ do
    $S \leftarrow L[p] + R[p] - m[p]$;
    if $S > bestSum$ then
        bestSum $\leftarrow S$; ;
        bestPeak $\leftarrow p$
$a[bestPeak] \leftarrow m[bestPeak]$;
for $i = bestPeak - 1$ to $1$ do
    $a[i] \leftarrow \min(m[i], a[i+1])$
for $i = bestPeak + 1$ to $n$ do
    $a[i] \leftarrow \min(m[i], a[i-1])$
return $a$;
```

(c) Complexity Analysis

- For creating $L[i]$ or $R[i]$ we push/pop each index at most once: $O(n)$ time.
- Choosing the best peak: $O(n)$.
- Reconstruction of a: $O(n)$.

Thus the overall time complexity is

$$O(n),$$

with $O(n)$ extra space for arrays L, R, and the output a.
End of Solutions.