ESO207A: Data Structures and Algorithms

Theoretical Assignment 2 Solutions

Question 1. The Archive Sorting Ritual (30 points)

Solution Sketch.

(a) A sorted array of size n triggers exactly one call to ArchiveSort. Each time we deliberately make a previously sorted segment unsorted, the algorithm splits into two recursive calls, increasing the total call count by 2. Thus a valid target k must be odd and satisfy $1 \leq k \leq 2n-1$.

We begin with $A = [1, 2, \ldots, n]$ and perform exactly $(k-1)/2$ such "splits". To create a split on a segment $[l, r]$, we swap the middle two elements $A[mid]$ and $A[mid+1]$, ensuring the segment is no longer sorted. We traverse segments in preorder and apply a swap whenever the global counter `swaps_needed` remains positive.

Algorithm 1 ConstructPermutation(n, k)

```
if k is even or k > 2n − 1 or k < 1 then
    return "No solution"
end if
A ← [1, 2, ..., n]
swaps_needed ← (k − 1)/2
Generate(A, 0, n − 1, swaps_needed)
return A
```

Algorithm 2 Generate$(A, l, r,$ `swaps_needed`$)$

```
if swaps_needed = 0 or l ≥ r then
    return
end if
mid ← ⌊(l + r)/2⌋
swap(A[mid], A[mid + 1])
swaps_needed ← swaps_needed − 1
Generate(A, l, mid, swaps_needed)
Generate(A, mid + 1, r, swaps_needed)
```

(b) A sorted segment yields 1 call; an unsorted swap forces 3 calls for that segment and its children, increasing the total by exactly 2. Each swap therefore increases the total call count by 2, so performing $(k-1)/2$ swaps yields exactly

$$1 + 2 \cdot \frac{k-1}{2} = k$$

calls. The base case $k = 1$ is trivial, and the inductive increase follows from the preorder traversal.

(c) Initialization takes $O(n)$. The recursion visits each nontrivial segment once, forming a merge-sort–like binary tree of $O(n)$ nodes. Each node does $O(1)$ work. Total time: $O(n)$.

Question 2. **Royal Guard Deployment** (30 points)

Solution Sketch.

(a) We compute, for every node v, two values:

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

These give the maximum strength in the subtree rooted at v when v is chosen or excluded. A postorder traversal computes both quantities in linear time. The final answer is max(inc[root], exc[root]).

Algorithm 3 Compute(v)

```
1: if v is null then
2:    return (0, 0)
3: end if
4: (Linc, Lexc) ← Compute(v.left)
5: (Rinc, Rexc) ← Compute(v.right)
6: v.inc ← v.val + Lexc + Rexc
7: v.exc ← max(Linc, Lexc) + max(Rinc, Rexc)
8: return (v.inc, v.exc)
```

To recover one optimal deployment, we perform a top–down traversal: if the parent is chosen, the child cannot be; otherwise the child is selected iff $\text{inc}[v] \geq \text{exc}[v]$.

Algorithm 4 Reconstruct(v, parent_chosen)

```
1: if v is null then
2:    return
3: end if
4: if parent_chosen then
5:    Reconstruct(v.left, false)
6:    Reconstruct(v.right, false)
7: else
8:    if v.inc ≥ v.exc then
9:       select v
10:      Reconstruct(v.left, true)
11:      Reconstruct(v.right, true)
12:   else
13:      Reconstruct(v.left, false)
14:      Reconstruct(v.right, false)
15:   end if
16: end if
```

(b) Both phases run in linear time.

- **Compute**: postorder traversal visits each of the n nodes once and does $O(1)$ work per node.
- **Reconstruct**: preorder traversal also visits each node once with $O(1)$ work.

Thus the overall time complexity is

$$T(n) = O(n) + O(n) = O(n).$$

Question 3. As Pretty As It Gets! (40 points)

Solution Sketch.

(a) The constraints imply that the final height profile must be unimodal: non-decreasing up to a peak p and non-increasing afterwards. For any candidate peak p, we construct:

$$a_p = m_p,$$

$$a_i = \min(m_i, a_{i+1}) \text{ for } i = p-1, \ldots, 1,$$

$$a_i = \min(m_i, a_{i-1}) \text{ for } i = p+1, \ldots, n.$$

This yields the maximum valid array for the fixed peak. Compute its sum S_p and choose the best p.

Brute force tries all $p \in \{1, \ldots, n\}$. Each trial takes $O(n)$ work, so the total time is:

$$O(n^2).$$

(b) Define for each p:

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

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

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

Thus S_p is the optimal sum if p is the peak. Both L and R are classic "sum of subarray minimums" arrays and can be computed in $O(n)$ using monotonic stacks.

Correctness (sketch). For any fixed peak, the greedy fill rule maximizes all a_i under unimodality. $L[p]$ and $R[p]$ compute exactly the sums of subarray minima ending/starting at p, so S_p is the optimal sum for peak p. Maximizing S_p gives the globally optimal skyline.

(c) Each of the three major phases runs in linear time:

1. Computing L is $O(n)$: every index is pushed/popped at most once from the monotonic stack. 2. Computing R is $O(n)$ by the same argument. 3. Peak search ($O(n)$) and reconstruction ($O(n)$) add linear work.

Total:

$$O(n) + O(n) + O(n) = O(n).$$

Algorithm 5 BestSkyline($m[1..n]$)

initialize empty stack
for $i = 1$ to n **do**
    {Compute L} **while** stack nonempty and $m[\text{top}] \geq m[i]$ **do**
        pop stack
    end while
    $pse \leftarrow$ (stack empty ? 0 : top)
    $L[i] \leftarrow (pse == 0 \ ? \ 0 : L[pse]) + m[i] \cdot (i - pse)$
    push i
end for
clear stack
for $i = n$ downto 1 **do**
    {Compute R} **while** stack nonempty and $m[\text{top}] > m[i]$ **do**
        pop stack
    end while
    $nse \leftarrow$ (stack empty ? $n + 1$: top)
    $R[i] \leftarrow (nse == n + 1 \ ? \ 0 : R[nse]) + m[i] \cdot (nse - i)$
    push i
end for
bestIdx $\leftarrow 1$, bestSum $\leftarrow -\infty$
for $i = 1$ to n **do**
    total $\leftarrow L[i] + R[i] - m[i]$
    if total $>$ bestSum **then**
        bestSum $\leftarrow$ total; bestIdx $\leftarrow i$
    end if
end for
$a[\text{bestIdx}] \leftarrow m[\text{bestIdx}]$
for $i =$ bestIdx $- 1$ downto 1 **do**
    $a[i] \leftarrow \min(m[i], a[i + 1])$
end for
for $i =$ bestIdx $+ 1$ to n **do**
    $a[i] \leftarrow \min(m[i], a[i - 1])$
end for
return $a[1..n]$