PSO 2 — Week 11: Solutions

CS381, Fall 2026

Problem 1

```
Skyline(buildings):
    if there is only one building, [l, r] with height h: return [(l, h), (r, 0)]
    split the buildings into two halves, in any way
    return Merge(Skyline(first half), Skyline(second half))

Merge(S1, S2):
    y1 := 0;  y2 := 0;  last := 0;  out := empty list
    while S1 or S2 has breakpoints left:
        x := the smaller of the next x-coordinates in S1 and S2
        if the next breakpoint of S1 is at x: take it, and set y1 to its height
        if the next breakpoint of S2 is at x: take it, and set y2 to its height
        if max(y1, y2) != last: append (x, max(y1, y2)) to out;  last := max(y1, y2)
    return out
```

The merge. Every building is in one of the two halves, so at each x the skyline of all the buildings is the higher of the two half-skylines. Merge walks through the breakpoints of both lists in increasing order of x, as in merge sort. The variables y_1 and y_2 hold the current heights of the two half-skylines, which change only at their breakpoints, so between two consecutive breakpoints of either list the higher of the two stays the same. Merge recomputes it at each breakpoint and outputs a new breakpoint only when it changes, so the output has no redundant entries. When both lists have a breakpoint at the same x, both are taken before the new height is computed.

Correctness. By induction on the number of buildings. A single building's skyline is $(\ell, h), (r, 0)$. If the recursive calls return the skylines of the two halves, the merge returns the higher of the two at every x, which is the skyline of all the buildings. □

Time. The height of a skyline can change only where some building starts or ends, so the skyline of m buildings has at most $2m$ breakpoints. Merging the skylines of the two halves of n buildings therefore handles at most $2n$ breakpoints and takes $O(n)$ time. The recurrence $T(n) = 2T(n/2) + O(n)$, as for merge sort, gives $T(n) = O(n \log n)$.

The same bound can be reached without divide and conquer, by sweeping from left to right and keeping the buildings that cover the sweep position in a heap ordered by height.

Problem 2

(a) Next greater element

```
S := empty stack of indices
for j = 1 to n:
    while S is not empty and A[top(S)] < A[j]:
        next(pop(S)) := j
    push j onto S
for each index i still on S: next(i) := none
```

Invariant. Just before index j is processed, the stack holds exactly the indices $i < j$ whose $\text{next}(i)$ has not been found yet — those with no element larger than $A[i]$ among positions $i+1, \ldots, j-1$ — and their values decrease from the bottom of the stack to the top. That ordering of the values is where the name "monotonic stack" comes from.

Correctness. Suppose the invariant holds when j arrives. For an index i on the stack, nothing between i and j is larger than $A[i]$, so $\text{next}(i) = j$ exactly when $A[i] < A[j]$. Because the values on the stack decrease from bottom to top, the indices with $A[i] < A[j]$ form a block at the top, and that block is exactly what the loop pops: it stops at the first index whose value is larger than $A[j]$, and everything below that one is larger still. Every index left on the stack then has a value larger than $A[j]$, so pushing j keeps the values decreasing, and j belongs on the stack because nothing after it has been seen yet. So the invariant holds for $j+1$. The indices left on the stack at the end have no larger element anywhere to their right. □

Time. Bounding each iteration separately fails, since one iteration may pop many indices. Count pops over the whole run instead: each index is pushed exactly once, so it is popped at most once, and the loop's test therefore succeeds at most n times in total; it fails exactly once per iteration. The total is $O(n)$. Graham's scan, after its sort, is bounded by the same argument: every point is pushed once, so it can be popped at most once.

(b) Largest rectangle in a histogram

The key fact. Take a largest rectangle, and say it covers bars ℓ through r. Its height is at most the shortest of $h_\ell, \ldots, h_r$, and it must be exactly that, or we could make it taller. Let i be a shortest bar in that range, so the rectangle has height h_i. The rectangle also cannot be widened: if bar $\ell - 1$ existed with $h_{\ell-1} \geq h_i$, it could extend one unit to the left. So bar $\ell - 1$ is shorter than bar i or does not exist, and since every bar from ℓ to i is at least as tall as bar i, bar $\ell - 1$ is the nearest bar to the left of i that is strictly shorter than bar i. Likewise bar $r + 1$ is the nearest strictly shorter bar to the right of i.

So for each bar i let

$$L(i) = \text{the nearest index left of } i \text{ whose bar is strictly shorter than bar } i \text{ (0 if none)},$$
$$R(i) = \text{the nearest index right of } i \text{ whose bar is strictly shorter than bar } i \text{ } (n+1 \text{ if none}),$$

and return $\max_i h_i\big(R(i) - L(i) - 1\big)$. Each candidate is a genuine rectangle, since every bar strictly between $L(i)$ and $R(i)$ is at least as tall as bar i; and by the key fact, a largest rectangle is among the candidates.

Computing L and R in $O(n)$. $R(i)$ is the first bar to the right of i that is *shorter*, so it is part (a) with the comparison reversed: scan left to right, and pop an index while its bar is strictly taller than the current bar, recording the current index as its R. L is the same scan run from right to left. Each scan is $O(n)$ by the counting argument of (a), and so is taking the maximum.

The tie rule matters. Nothing above needed distinct heights. If several bars in $\ell, \ldots, r$ share the minimum height, each of them gets the same L and R, so the same rectangle is found more than once, which is harmless when taking a maximum. What does matter is that both comparisons are strict. With heights $1, 1$ the answer is 2, but if $L(i)$ and $R(i)$ were the nearest bars of height *at most* h_i, each bar would stop at the other, both widths would come out as 1, and the algorithm would report 1.