Code Jam 2019 - Round 3

Analysis: Pancake Pyramid

We can make a couple of useful observations at the outset. First, if we have an interval of length 1 or 2, we do not need to add any pancakes for it to have the pyramid property, so we can ignore the restriction of length ≥ 3 in the problem. Second, for any interval, the "peak" in the optimal answer is the largest stack in the original interval (we leave this for you to think about). If there are multiple largest stacks in an interval, we will take the leftmost largest stack as the peak.

$O(S^3)$ — Too slow

For each of the ((**S** + 1) choose 2) intervals, determine where the peak will be located once the interval is turned into a pyramid. Once we know where the peak will be, we have two smaller problems: we need a non-decreasing sequence to the peak's left and a non-increasing sequence to the peak's right. To compute how many pancakes we need in the non-decreasing interval, we may simply sweep from the leftmost point and add pancakes until no stack in the interval to the left of the i-th stack is strictly taller than the i-th stack. By maintaining the running maximum as we sweep, we can compute the number of needed pancakes needed per interval in $O(\mathbf{S})$ operations. Since there are $O(\mathbf{S}^2)$ intervals, this algorithm requires $O(\mathbf{S}^3)$ operations in total.

$O(S^2)$ — Test Set 1

The ideas above lay the framework for a quicker solution. Instead of independently recomputing the number of pancakes needed to make an interval non-decreasing (or non-increasing), we can use the results from other intervals. Say we know the index of largest stack of pancakes in the range [L, R] (call this index M[L, R]) and the smallest number of pancakes needed to make the interval [L, R] into a non-decreasing sequence (call this number X[L, R]). We can compute both M[L, R+1] and X[L, R+1] in O(1) time since the height at M[L, R+1]=max($\mathbf{P_{M[L,\ R]}}$, $\mathbf{P_{R+1}}$) and X[L, R+1] = X[L, R] + ($\mathbf{P_{R+1}}$ - $\mathbf{P_{M[L,\ R+1]}}$). Similarly, we can store Y[L, R], which is the smallest number of pancakes needed to make the interval [L, R] into a non-increasing sequence.

For any interval [L, R], the smallest number of pancakes needed to turn the interval into a pyramid is simply X[L, M[L, R]] + Y[M[L, R], R]. The precomputation takes $O(\mathbf{S}^2)$ time and memory and the second step uses O(1) time per interval to compute the answer. Thus, in total, this is $O(\mathbf{S}^2)$.

O(S log S) — Test Set 2

The above strategy will be too slow and require too much memory to handle the larger bounds. For this test set, we will still use the same underlying idea of needing the number of pancakes to make an interval non-decreasing or non-increasing. But instead of computing X and Y, we will compute cumulative values: define X'[L, R] = X[L, R] + X[L+1, R] + ... + X[R, R] and Y'[L, R] = Y[L, L] + Y[L, L+1] + ... + Y[L, R]. Now, instead of focusing on the left and right endpoints, we will base our strategy on the peaks of the intervals.

Initially, we do not know the value of X' or Y' for any interval. In our analysis, we will assume that we only know the X' and Y' values for maximal intervals. That is, if two intervals are side-by-side, we will merge them (we will never have intersecting intervals). For example, if we know

X'[L, k] and X'[k+1, R], we will merge these together into X'[L, R] and forget about X'[L, k] and X'[k+1, R]. The full process of how to merge is explained below. In particular, this means that any given stack is in at most one known interval for X' and one known interval for Y'.

We will process the peaks from smallest to largest. When we process the i-th stack, we are only interested in intervals that have stack i as their peak. If X'[L, i-1] is computed for some value of L, then L must be the furthest left index such that P_L, P_{L+1}, ... , P_{i-1} are all less than P_i (since we are processing the stacks from smallest to largest). Similarly, if Y'[i+1, R] is computed for some value R, then R must be the furthest right index such that P_{i+1}, P_{i+2}, ... , P_R are all at least P_i. If such L and R exist, then we can compute the number of pancakes needed over all intervals that have i as their peak:

X'[L, i-1] * (R - i + 1) + Y'[i+1, R] * (i - L + 1)

Note that if we don't know X'[L, i-1] for any L, then $\mathbf{P_{i-1} \geq P_i}$, so i cannot be a peak with any interval that includes both i-1 and i. The answer for those intervals will be computed later when we consider stack i-1 as the peak (and likewise for stack i+1 if we do not know Y'[i+1, R] for any R). In these cases, we may use X' = 0 (or Y' = 0). Note that since our intervals are maximal and we are computing from smallest to largest, $\mathbf{P_{L-1} \geq P_i}$ (similarly, $\mathbf{P_{R+1} > P_i}$).

Now we want to merge X'[L, i-1], X'[i+1, R] and stack i into X'[L, R]. We will do this in two steps. First, note that X'[L, i] = X'[L, i-1]: since we are processing the stacks from smallest to largest, P_i can be freely added as the right endpoint of any non-decreasing sequence in this range. Now let's merge X'[L, i] and X'[i+1, R]. Observe that X'[L, R] sums over intervals that end at the R-th stack. If an interval starts in the range [i+1, R], then it is already counted in X'[i+1, R]. If an interval starts in [L, i], then we can start with some sequence in [L, i], but since $\mathbf{P_i}$ is the peak, every value on the right must be exactly $\mathbf{P_i}$. The number of pancakes needed to bring every value in [i+1, R] up to $\mathbf{P_i}$ can be computed in O(1) time using cumulative sums. Thus, the full merge is:

$$X'[L, R] = X'[i+1, R] + (X'[L, i-1] + \mathbf{P_i} \times (i-L+1) \times (\mathbf{P_{i+1}} + ... + \mathbf{P_R}))$$

The Y' values can be computed similarly. In terms of complexity, we need to sort the stacks at the beginning in O(**S** log **S**) time and the remaining steps take constant time per peak, so O(**S**) overall. This means the algorithm takes O(**S** log **S**) time.

O(S)

Although the O(**S** log **S**) solution is fast enough to solve test set 2, an O(**S**) solution is also possible! We present a sketch of the idea here, which can be read independently of the solution above.

To make things easier, we pretend that the stacks all have different heights. Each time we compare the heights of two stacks of identical height, we break the tie by assuming that the stack with the larger index is higher.

Let us think through the solution starting from the end. For each stack, we want to compute the pyramidification cost for all the ranges in which this stack is the highest; in such cases, it will be the peak of the pyramid. Then we can sum all of those values for all of the stacks, and that will be our overall result.

In order to compute those pyramidification costs, we can compute the following for each stack *s*:

- The nearest higher stack to the left of *s* (possibly an infinitely high "guard" stack appended to the beginning of the sequence); call this the "left blocker". Let D_L be the absolute distance (in stacks) from *s* to the left blocker.

- The nearest higher stack to the right of *s* (or guard stack) stack added after the sequence); call this "right blocker". Let D_R be the absolute distance (in stacks) from *s* to the right blocker.
- The pyramidification cost for all the ranges that end with *s* and in which *s* is the highest; call this "left pyramidification cost" C_L.
- The pyramidification cost for all the ranges that start with *s* and in which *s* is the highest; call this "right pyramidification cost" C_R.

Then the pyramidification cost for *s* can be calculated as $C_L \times D_R + C_R \times D_L$.

Now, how can we compute C_L and D_L for each stack? (The solution is analogous for C_R and D_R; the only major difference is that the inequality used to compare stacks is strict on one side and non-strict on the other, due to tie-breaking.)

We can traverse the stacks from left to right, keeping a Stack structure (capital S to avoid confusion) X that remembers the longest decreasing sequence of stacks ending on the current stack. We iterate through the stacks and when seeing a stack s we consume from X all stacks that are lower than s, adding their contributions to the current left pyramidification cost. Let t' = s at the beginning, and later t' = the previously consumed stack. The contribution of a consumed lower stack t is the number of pancakes missing between t and t' (calculated in constant time, if we precompute the cumulative sum of stack sizes) multiplied by the distance to the left blocker of t'. The left blocker of s is the first stack we can't consume from X, because it's higher than s. Once we're done with s, we add s to X and keep going.