Kick Start 2020 - Round C

Analysis: Perfect Subarray

For test set 1, we can use the brute force approach to generate all subarray sums, check if each one is a square and return the total count. This would be enough to pass all the test cases under the time complexity.

For test set 2, looking at the problem constraints, we can estabilish that the largest subarray sum possible across all testcases would be **N***MAX_A where MAX_A is the largest element in **A**. Therefore, we can precompute all squares ≤ **N***MAX_A. This amounts to √(**N***MAX_A) squares. Let's call this S[].

First, let's define Res[] where Res_i stores the number of subarrays ending at index i with subarray sum that is a perfect square.
Note: sum(**A**[L....R]) = sum(**A**[0.....R]) - sum(**A**[0....L-1]) for 0 < L ≤ R ≤ **N**..
Next, define an array P[] that keeps count of the number of indices i such that **A**[0...i] amount to a specific prefix_sum. i.e., P[prefix_sum] should give us the number of indices i such that sum(**A**[0...i])=prefix_sum. However, we could have negative prefix_sum values and hence P[prefix_sum] could be an invalid lookup. To resolve this, instead of mapping a prefix_sum to P[prefix_sum], we map it to P[prefix_sum + offset], where offset = min(sum(A[0.....i]), 0)*-1 for 0 ≤ i < **N**. i.e., The minimum among the **N**+1 (+1 for the empty prefix) prefix_sum values possible which can be computed with a single pass over **A**. Note that the offset is at least 0.

Next, we iterate the **A** left to right, while maintaining the sum of elements seen so far - let's call that prefix_sum. Now, at every i-th index, we ask the question, *How many subarrays end at i and have the subarray sum which is also a square*?
To answer this, we iterate S[] and for each square S_k, we add P[(prefix_sum-S_k) + offset] to Res[i]. Why so? - P is built as we iterate **A** and hence, at a certain index i, P holds the mapping of {prefix_sum, count} where count is the number of indices j (< i) such that sum(A[0....j])=prefix_sum. Therefore, P[(prefix_sum-S_k) + offset] holds the number of indices such that sum(A[j...i])=S_k.
We also increment the count of P[prefix_sum+offset] by 1 to record that sum(**A**[0....i])=prefix_sum. Finally, summing up all values Res[] would give us our answer.

Since we traverse **A** once, iterate S[] for every index i and lookup in P[] is O(1), the total time complexity for this solution is O(**N***√(**N***MAX_A)).

Appendix

A subtle observation and a potential improvement is to early exit on iteration of S[] at every stage. As mentioned earlier, we check P[(prefix_sum-S_k) + offset] and notice that at some point (prefix_sum-S_k) + offset could become < 0 which indicates not only that accessing P would be invalid, but also that S_k is too large to be obtained from all elements upto the current index i. We can use this criteria as a way to early exit the iteration on S[]. The asymptotic time complexity would remain the same, but would be slightly faster in run-time.

Next, instead of using an array with an offset for lookup of prefix sums , we could use a normal map, which would remove the need of an offset, but adds the cost of lookup that would take logarithmic time instead of the O(1). This solution may also be accepted if written efficiently and the time comoplexity would be O(**N***log(**N**)*√(**N***MAX_A)).