Kick Start 2018 - Round D

Analysis: Candies

Candies: Analysis

To make our discussion easier, for each range (L, R), let us define SS(L, R) to be the sum of sweetness of the candies from index L to R (inclusive), and SO(L, R) to be the number of odd candies from index L to R (inclusive). Our problem is equivalent to finding a range (L, R) such that:

- SO(L, R) ≤ **O**.
- SS(L, R) ≤ **D**.
- SS(L, R) is maximized.

Small dataset

We will use the technique of **two pointers** to solve the Small dataset. For each value of L from 1 to **N**, we would like to find the value of R such that the value of SS(L, R) is maximized while observing the constraints stated above. Since the candies have non-negative sweetness in the Small dataset, for every L, we simply have to find the rightmost index R that satisfies SO(L, R) ≤ **O** and SS(L, R) ≤ **D**.

For any fixed L', suppose we have identified that R' is the rightmost index that satisifes SO(L', R') ≤ **O** and SS(L', R') ≤ **D**. For L'+1, note that S(L' + 1, R') necessarily satisfies SO(L' + 1, R') ≤ **O** and SS(L' + 1, R') ≤ **D**. Thus, we do not need to iterate through the entire array to find the value of R that maximizes SS(L'+1, R). We simply have to iterate from R' + 1 to **N** to find the largest value of R that still satisfies our constraints. This dramatically reduces the runtime of the algorithm, as we only iterate the array once to find the optimal value of R for each L (Hence the name two pointers — the pointers L and R only iterate over the array once). Once we obtain the optimal sweetness for each L, we can simply take the maximum and return as the answer. Total runtime of the solution is O(**N**).

Large dataset

Since $\mathbf{S}_i$ may be negative in the Large dataset, for each L, we can no longer assume that the rightmost index that satisfies the constraints will give us the optimal value for SS(L, R). However, we can still use the above sliding window technique to identify the rightmost index that still satisfies the oddness constraint. For any fixed L', suppose we have identified that R_O' is the rightmost index that satisfies the oddness constraint. That is to say, we have SO(L', R_O') ≤ **O** and SO(L', R_O' + 1) > **O**. Now we need to find the index R in the range [L', R_O'] that maximizes the value of SS(L', R), while observing the constraint SS(L', R) ≤ **D**.

To do so, we shall precompute the prefix sum of the sweetness of the candies (i.e. the values of SS(1, 1), SS(1, 2), ..., SS(1, **N**)). Notice that SS(L, R) = SS(1, R) - SS(1, L - 1). Thus, finding the value of R that maximizes S(L', R) is equivalent to finding the value of R that maximizes SS(1, R) - SS(1, L' - 1), which is the same as finding **the value of R that maximizes the prefix sum SS(1, R) while observing SS(L', R) ≤ D.**

In order to identify the value of R quickly, we will need a data structure that supports adding integers, removing integers, and finding the largest integer smaller than a queried integer. A **balanced binary search tree** (e.g. C++ multiset) can perform each of these operations in logarithmic time. As we iterate over the value of L, we will add or remove the values of SS(1, R) to the tree such that the tree only contains prefix sums for the indices in the range $[L, R_O]$. Then, we will query for the largest prefix sum such that $SS(1, R) \le SS(1, L - 1) + D$. This allows us to compute the maximum sweetness for a subarray that starts from L, and we may then compute the overall maximum sweetness across all subarrays.

The time needed to preprocess all prefix sums and values of R_O is O(**N**). Subsequently, each prefix sum may only be added or removed at most once from our data structure. In addition, we will only query the data structure once for each value of L. Using a balanced binary search tree, this gives us a total runtime of O(**N** log **N**).