Code Jam 2019 - Round 1B

Analysis: Fair Fight

Test set 1

Let us call a pair (L, R) *fair* if choosing it produces a fair fight. Notice that since there are only 100 entries, there are at most 100*101/2 = 5050 possible intervals. For each interval, we may simply search for the maximum value is in each array via a linear search and check if those maximum values are close enough.

Test set 2

Notice the random choice of sword in case of ties does not change whether (L, R) is fair or not, so we can further assume that, if there are ties, they are broken by choosing the sword with smallest index. To simplify the write-up below, we assume that each $\mathbf{C}_i$ is distinct.

Let us consider a sub-problem: for each sword i that Charles can choose, how many fair intervals (L,R) are there which Charles chooses sword i? Let us call this value F_i. The answer to the original problem is simply the sum of the F_is. For each interval (L,R), there are three properties we are concerned with:

- (P1) Charles chooses sword i. That is, $L \le i \le R$ and $\max(\mathbf{C}_L,\mathbf{C}_{L+1},...,\mathbf{C}_R) = \mathbf{C}_i$.
- (P2) Charles' sword is *good enough*. That is, $\max(\mathbf{D}_L,\mathbf{D}_{L+1},...,\mathbf{D}_R) \le \mathbf{C}_i + \mathbf{K}$.
- (P3) Charles' sword is *too good*. That is, $\max(\mathbf{D}_L,\mathbf{D}_{L+1},...,\mathbf{D}_R) < \mathbf{C}_i - \mathbf{K}$.

So we have F_i = (# of intervals that satisfy P1 and P2) - (# of intervals that satisfy P1 and P3). These two quantities are computed very similarly since they just have a different bound on the right-hand side of the inequality in P2 and P3. We explain below just how to compute (# of intervals that satisfy P1 and P2) and leave computing (# of intervals that satisfy P1 and P3) to the reader.

Note that if an interval (L,R) satisfies P2, then any subinterval of (L,R) also satisfies P2. Similarly, if (L,R) satisfies P1, then any subinterval of (L,R) that still contains i also satisfies P1. Thus, we are really only interested in how far left L can go with R=i (and how far right R can go with L=i). One option is to do a linear search for how far left L can go, but this is too slow. Instead, we binary search for how far away the left-endpoint is. A left-endpoint is too far left if P1 or P2 are no longer true. Otherwise, the left endpoint can be pushed further left. Once we know the furthest left L can go (let this index be L_i) and the furthest right R can go (let this index be R_i), then (# of intervals that satisfy P1 and P2) = $(i - L_i + 1) \times (R_i - i + 1)$. Calculating the maximum element in a range can be computed efficiently using a range minimum (maximum) query-type data structure in O(1) time per query, meaning O(log **N**) total time for the binary search.

In terms of time complexity, for each i, we need to perform 4 binary searches, which take O(log **N**) time each, for an overall O(**N** log **N**) total time. Setting up the range maximum query data structure takes an additional O(**N** log **N**) time, yielding an overall O(**N** log **N**) time for the full algorithm. There are solutions that can compute all of the required L_i and R_i values (defined above) in O(**N**) total time by doing a couple of clever sweeps of the data to count the number of unfair intervals, but this was not needed for this problem.