Kick Start 2019 - Round A

Analysis: Contention

The number of possible orderings of the requests is Factorial(**Q**), which would not be fast enough for either of the test sets. We can observe that for a chosen ordering of the requests, the number of seats that the system books in the last request does not depend on the ordering of the previous **Q** - 1 requests. So, we could start by finding the request to be processed last and move backwards towards the earlier requests. The answer is the minimum seats booked across the **Q** steps.

Another observation is that at every step, we can always choose this last request greedily from the remaining set: the one where we can book the maximum number of seats. An intuitive proof of why this works would be noticing that our final answer is non-increasing over the **Q** steps. Now, we can use exchange argument to prove this observation since choosing a request with lesser seats booked would not give us a better answer.

Test set 1 (Visible)

A naive implementation would be of the order O(**N** × **Q**), where we recalculate number of seats for remaining requests in O(**N**) using sweep line algorithm for each of the **Q** steps. We can speed this up with another observation: the number of unique ranges that the requests cover is at max 2 * **Q**, which would make our solution run in O($\mathbf{Q}^2$) for every test case.

Test set 2 (Hidden)

Let us try to speed up the slow process of recalculating number of seats we can book at every step For every seat, let us denote the *value* of a seat as the number of remaining requests trying to book this seat. Everytime the value of a seat drops to 1, we increase the number of seats we can book for the corresponding request containing this seat booking.

Since requests are represented by an interval, we can use an interval tree to support the operations of removing an interval after the initial construction of the tree. Each node of the interval tree stores the set of intervals that cover it, and also the minimum value of any seat in it's range. Whenever a remove operation makes the minimum value become one, we walk down the tree to find the seats that became one, and walk back up the tree to find out which interval is the only interval that now covers that seat. We now set that seat's value to infinity so that we don't process it again. This happens only once per seat, for a total amortised cost of O(**N**log**N**). In total this algorithm is O(**N**log(**Q**+**N**)), if you use a constant time set (like a hashset).