Kick Start 2021 - Round F

Analysis: Festival

Test Set 1

We can consider each attraction as an interval that covers some range of days and has some *happiness rating*. Let us denote *happiness rating* of an interval as HR.

At first we sort the intervals according to their HRs, from higher to lower. Then we will iterate through each day and consider that day as the day we will attend the festival. Let us denote this day as d_i. And let us denote the total happiness of upto $\mathbf{K}$ intervals on this day as c_i.

Let us set c_i to 0. Now we will iterate through the sorted intervals and check if d_i is included in the current interval. If yes, then we add the HR of that interval to c_i. If no, we continue to the next interval. If the number of intervals taken into account reaches $\mathbf{K}$, we stop calculating c_i and compare it to the current maximum answer and store it as maximum if it is greater. If we iterate through all the intervals and the number of intervals taken is less than $\mathbf{K}$, we still compare c_i to the current maximum answer and store it as maximum if it is greater.

The answer is the maximum among all c_is.

The time complexity of this solution is $O(\mathbf{N} \times \log \mathbf{N} + \mathbf{D} \times \mathbf{N})$.
The space complexity (without considering the input) of this solution is $O(1)$ as no extra memory is used.

A pseudocode would look something like this:

```
ans = 0
sort(intervals) // from higher to lower
for d_i = 1 to D:
  c_i = 0
  cnt = 0
  for j = 1 to N:
    if (cnt == K): break
    if (s_j <= d_i <= e_j):
      c_i += h[d_i]
      cnt++
  ans = max(ans, c_i)
return ans
```

Test Set 2

Now the brute force will not work. We can still iterate through each day, but now we cannot check each interval to see if this day is in that interval or not. Rather for each day, we will update the events taking place on that day efficiently and then calculate the total HR for that day. We will consider the ending days as the ($\mathbf{e_i}$ + 1)-th days, because the $\mathbf{e_i}$-th day is included in the interval. So now we will consider the ($\mathbf{e_i}$ + 1)-th day as the ending day for each interval. For example, if an interval ended on day 6 previously, now we will consider that it ends on day 7.

For each day, we will store the intervals that either start or end on that day.

We will use segment trees for our solution. Each leaf node of the tree will be an interval and the intervals will be sorted according to their HRs, from higher to lower. Initially every interval will have a value of 0. When an interval becomes active, we will update that index with its HR. And when an interval becomes inactive, we will update that index with 0 again. We need to query the sum of any range in the trees. We also need another operation. At any moment we need to find the index of the $\mathbf{K}$-th maximum (HR wise) active interval. If there are less than $\mathbf{K}$ active intervals, we will return the index of the rightmost/last active interval. We can use another segment tree for this.

Now we will iterate through the days.

If the current day is a starting day for an interval, it means this interval has become active. We will update its index with its HR in one tree and will make it active in the other one.
If the current day is an ending day for an interval, it means this interval has become inactive. We will update its index with 0 in one tree and will make it inactive in the other one.
We will do this for each interval for which the current day is a starting day or an ending day.

On the days that we update the segment trees, after updating, we will query for the index of the $\mathbf{K}$-th maximum active interval in the second tree. Let us assume this index is idx. And then in the first tree, we will query the sum from the 0-th index to index idx. We will store this sum in the current answer if the current answer is smaller.

The answer is the final value in the current answer.

The time complexity of building a segment tree with n elements is $O(n)$ and for updating and querying the time complexity is $O(\log n)$ for each operation.
So the time complexity of this solution is $O(\mathbf{N} + \mathbf{D} + \mathbf{N} \times \log \mathbf{N})$.
And the space complexity of this solution is $O(\mathbf{N})$.

Alternative Solution

We will discuss another solution using multisets which is available in C++ standard template library (STL). It can also be implemented using a binary search tree in any language.

We will use two multisets, one to keep track of the $\mathbf{K}$ active intervals with higher HRs, and the other one to keep track of the rest of the active intervals.

Now we will iterate through the days again.

If the current day is a starting day of an interval, then if the number of active intervals in the first multiset is less than $\mathbf{K}$, we insert this interval in the first multiset. Otherwise, if the interval with the smallest HR has a greater HR than this interval, then we insert this interval in the second multiset. And if it is less or equal, then we remove the smallest from the first multiset and insert the removed interval into the second multiset. And then we insert the current interval in the first multiset.

If the current day is an ending day of an interval, then if this interval is in the second multiset, we simply remove it from there. Otherwise it is in the first multiset. So, we remove the current interval from the first multiset. Then we remove the interval with the largest HR from the second multiset and insert the removed interval in the first multiset.

We always maintain a variable sum which is the sum of the HRs of the intervals in the first multiset. While inserting/removing an interval in/from the first multiset, we have to update the value of the sum accordingly.

The answer is the maximum value of sum across the whole time.

The time complexity of accessing, inserting, removing elements from a multiset with n elements is $O(\log n)$ for each operation. Since there are $\mathbf{N}$ intervals and each interval will be handled 2 times and each handling will require $O(1)$ operations with the multisets, the overall complexity of all operations with multisets will be $O(\mathbf{N} \times \log \mathbf{N})$.
So, the overall complexity of this solution is: $O(\mathbf{D} + \mathbf{N} \times \log \mathbf{N})$.
And the space complexity of this solution is $O(\mathbf{N})$.