Analysis

Exponential solution

For very small constraints we can simulate the games with exponential complexity – we simply try all possible ways the game could go. Since on every turn a player can have many different choices, this solution runs very slowly. It would work for about $N \leq 10$, depending on the implementation.

Polynomial solutions

The main observation in the problem is the following statement:

The optimal choice for the player whose turn it is is to take the largest number in the multiset.

This statement is obvious, since taking any other number does not obstruct the other player in any way and only reduces the player's own score.

In such case, we can just simulate all K games.

Solution for $O(N^2 \times K)$

On every turn, we have to find the largest number in the multiset. We can do this by simply iterating through the multiset linearly. Since its size can be at most $O(N)$, the complexity for the simulation of a single game is $O(N^2)$ and the total complexity of the solution is $O(N^2 \times K)$.

Solution for $O(N \times K \times \log(N))$

We can improve our previous solution by keeping the elements that are in the multiset in some structure that allows to quickly find the maximum number, as well as to remove the largest element and to add new elements. There are many structures that support these operations with logarithmic complexity per operation. The competitors can implement a structure of their own or use one of the STL structures – for example *priority_queue* or *set/map*. The best complexity that can be obtained using a structure of this kind is $O(N \times logN)$ for the simulation of a single game and hence a total complexity of $O(N \times K \times \log(N))$.

Solution for $O(N \times K)$

In order to improve the efficiency of our simulation we have to make the following observation:

If in a certain moment the number that is being added to the multiset is the largest number in the multiset, then this number will be taken on the very next turn.

This statement follows directly from the optimal playing strategy. In such case, let us define:

count[x] = the amount of times we have the number x in the multiset at a certain moment

Since all numbers are up to N, the size of *count*[] is also N.

The optimal move at a certain moment is to take the largest x, such that *count*[x] > 0. Adding and removing numbers can be done in O (1) time by simply increasing or decreasing the respective value in *count*[] with 1. The algorithm to quickly simulate a single game is as follows:

We begin with a pointer $ptr = N$ and we want to have *count*[x] = 0 for every $x > ptr$ at any moment. To find the optimal choice at a certain moment we reduce the pointer until we reach *count*[*ptr*] > 0. When we are adding a new number A to the multiset, we have two options:

1) If $A > ptr$, then we remember that this number will be taken on the very next move. We don't have to update *count*[] at all.
2) If $A \leq ptr$ then we can simply add the number and find the optimal choice using the pointer.

Since we reduce the pointer at most N times and process every addition in constant time, the total amortized complexity for the simulation of a single game is O (N). Thus we get a total complexity of O ($N \times K$).