B: Hormiga Battle Royale

Lo primero que hay que notar en el problema es que para que en un rango una hormiga gane todas sus peleas, esta tiene que ser el máximo común divisor de todos los números de ese rango, entonces la pregunta es dado un rango ¿cuántos números de ese rango son iguales al máximo común divisor del rango?

Para responder esta pregunta con el segment tree vamos a tener dos valores guardados en cada nodo: el máximo común divisor del rango y la cantidad de números iguales al mcd en este rango. Claramente cuando el rango solo tiene un elemento xi estos valores son {xi,1}.

Para unir dos rangos calcular el nuevo mcd es fácil, solamente calculamos el mcd entre los dos mcd anteriores con la función __gcd de c++.
Ahora, para contar cuántas veces está el mcd en el nuevo rango tenemos que comparar los mcd de los rangos antiguos con el nuevo, si el mcd del rango antiguo es igual al nuevo ya sabemos cuantas veces se repite en ese rango, es el segundo valor de nuestro nodo así que sumamos eso a la cantidad de veces que aparece. Si no es igual al nuevo mcd, por propiedades de mcd, no aparece ninguna vez en este rango.

De esta forma construimos un segment tree que responde a la pregunta que hace el problema y podemos usarlo para responder a las consultas del problema.

D: Sapo y Sepo en el paseo DCC

Primero hay que darse cuenta de que k puede ser mucho más grande que el n, y que puede darse el caso que tengamos que darle varias vueltas al círculo de participantes antes de llegar al que debemos eliminar, es más eficiente solo considerar la última vuelta, para eso consideramos p como la cantidad de jugadores restantes, luego definimos k2 = k % p.
Ahora k2 es la cantidad equivalente de jugadores que debemos saltar.

Para este problema usamos la función SegmentTree<T>.search(int from, T val) que viene en la implementación de SegmentTree que les pasamos, lo que hace esta función es buscar el menor j tal que SegmentTree<T>.query(from, j) sea exactamente val, la implementación de esta función se aprovecha de la la estructura de árbol interna del SegmentTree para conseguir una complejidad de O(log n) y por eso se le llama “búsqueda sobre el árbol”, si no usáramos esta función y utilizaramos búsqueda binaria en su lugar el algoritmo sería O(n logˆ2 n) lo que es muy justo al tiempo límite.

Esto lo usaremos para encontrar eficientemente cual jugador debe ser eliminado, para lograr esto debes hacer que SegmentTree<T>.search(a, b) retorne la cantidad de jugadores restantes en el rango [a, b]. Para esto vamos a usar una suma como función merge y por cada jugador activo vamos a marcar un 1, y cuando un jugador sea eliminado lo vamos a actualizar a un 0.

Ahora solo queda iterar hasta que se hayan eliminado todos los jugadores, en cada iteración calculas el k2, encuentras la posición del jugador que se va a eliminar usando la función search y se actualiza. Notar que hay que considerar que no queden suficientes

jugadores a la derecha, en cuyo caso hay que dar la vuelta y hacer un search desde el comienzo. La complejidad de este algoritmo es O(n log n)