Candies

Subtask 1

Since all a_i are equal to 1, the answer for each query is $r_i - l_i + 1$, which is the length of the interval.

Subtask 2

Due to the fact that each friend receives the same set of candies as other friends, each friend's set is equal to the interval's set. Furthermore, to ensure that after giving candies to the first friend other friends could receive the same type of candies, set of candies on the interval's prefix must be equal to the set of elements after this prefix.

Hence, it can be concluded that greedily picking the first prefix of candies that satisfies the requirement above is a valid strategy. Let's call the process of picking the required prefix as "jump". Now, this idea can be implemented for this subtask.

Subtask 3

For each query, there can be two solutions depending on the number of distinct candies. If there is only a single type of candy, then it is the same as in subtask 1.

Otherwise, there can be only two types of candies on the interval. One can pre-calculate the closest position of the next distinct candy for each position, and use it to optimize solution of subtask 2. Now, it will work in O(ans), where ans is the answer for the query. However, it is too slow, so binary lifting could be used to answer queries in $O(log(n))$.

Subtask 4

If the length of query's interval is odd answer is equals to 1, because there would be some element that has unique occurrence. Otherwise, the answer could be 2 if and only if the set of candies on the first half of the interval is equal to the set of the second half. Counting number of distinct elements is a famous task that can be done in $O(n \cdot log(n) + q \cdot log(n)))$ by using segment tree of fenwick tree.

Subtask 5

Observe, that after giving candies to the first friend, we left with the same task but for the new interval. Also, since each friend's set of candies is equal to the interval's set the number of distinct candies is the same for both of them.

Now, define $prefAns_i$ as the answer for the prefix ending at position i. Using the greedy strategy, if we can find the closest left position j where number of unique candies on interval $(j, i]$ is the same as on $[1, i]$, then $prefAns_i = prefAns_j + 1$ if number of unique candies on interval $[1, j]$ is the same as $[1, i]$. Otherwise, it will be equal to 1. Number of unique candies for each prefix could be calculated in $O(n)$, and position j could be found in $O(log(n))$ by keeping the right-most occurrence of each candy in $std :: set$.

Subtask 6

We can use the same solution from subtask 2, but do a little modification for it. Since number of unique candies on the interval cannot be larger than 100, one can answer queries when number of unique candies on query is equal to K independently. It is enough to pre-calculate for each i the next closest position where number of unique candies will be equal to K, which can be done in $O(n)$ using two pointers, and do

dfs and binary search, or binary lifting to answer queries. As result, it will be done in $O(n \cdot D + q \cdot log(n))$, or $(n \cdot D \cdot log(n))$ depending on the realization, where D is number of distinct candies.

Subtask 7

We are going to use SQRT optimization. Suppose the number of unique candies on the query's interval is less that $\sqrt{n}$, then we can use the solution from subtask 6 to answer all those queries in $O(n \cdot \sqrt{n} + q \cdot log(n))$.

As for the other case, the answer can not be greater than $\sqrt{n}$, because each friend gets an interval with at least $\sqrt{n}$ elements.

Let's fix some R, and answer all of the queries that end in the position R. For current R we will maintain an array $next_i$. It will contain the next closest position where number of unique elements is the same as on the interval $[i, R]$ (this array stores information about the intervals of candies that our friends get). Using this array we can "jump"to the next interval and find an answer in no more than $\sqrt{n}$ "jumps".

However, we need to update this array when we increment R. To do so we need to observe that $next_i$ is going to change only for positions after the previous occurrence of a_{R+1}. For such positions we have to assign $next_i = R+1$. If we do this updates and keep track of unique numbers using SQRT decomposition, we can get $next_i$ and number of unique numbers on interval $[i, R+1]$ in $O(1)$. Overall we get a solution that works in $O(q \cdot \sqrt{n} + n \cdot \sqrt{n})$.

Subtask 8

To solve the last subtask we will use different approach. Let's solve it using Divide And Conquer algorithm. We define recursive function $solve(L, R)$, this function finds an answer for all queries i such that $l_i \leq M$ and $M < r_i$, where $M = [\frac{L+R}{2}]$. For the queries that lie in the left half of the $[L, R]$ we will recursively call $solve(L, M)$, for other queries we will use $solve(M + 1, R)$.

Now, we have to observe that part of the query that lies in the left side of $[L, R]$ is independent from the right side. It can be obtained this way: notice that if the set of $[l_i, M]$ is the same as the set of $[l_i, r_i]$ our greedy approach will make "jumps"independently from the right side of the query (similar applies for the right side). Additionally, if the set of $[l_i, M]$ is smaller than the set of $[l_i, r_i]$ we can ignore "jumps"from the left side, and use "jumps"from the right side.

Now we can solve problem independently on the suffixes of $[L, M]$ and the prefixes of $[M + 1, R]$. This problem is similar to the Subtask 5. For the last detail we have to store the last "jump"from the left side and the last "jump"from the right side, as well as the number of jumps, and the number of different elements on the $[l_i, M]$ and $[M + 1, r_i]$. It is easy to store this values by applying small changes to the solution of the Subtask 5, however, we can make it slightly faster by using "two pointers"instead of $std :: set$.

The last detail that is left is the last "jump"from the left side and the last "jump"from the right side. We have to merge this to "jumps"because sometimes they form a valid interval, in this case answer for the query should be incremented.