A. Reflecting

The critical observation is to see where we can reach through a series of reflections. Trying out some points, we conjecture that we can only reach positions that are an even distance away from both our x-coordinate and our y-coordinate.

Indeed, this claim is true! Let our current position be (x_1, y_1) and the point we are reflecting through be (x, y). Without loss of generality, let $x > x_1$ and $y > y_1$. We can see that we will end up in the point $(x_1 + 2(x - x_1), y_1 + 2(y - y_1))$. This is clearly an even distance away from both our x-coordinate and our y-coordinate.

It is also not hard to prove sufficiency either. We can simply reflect through the midpoint between our current location and our desired location to reach it.

Thus, we write a program that prints YES if the parities match, and NO otherwise. The runtime is $O(1)$ per test case.

B. Numbers on the Blackboard

Let's say Yunli already wrote all of the numbers from 1 to n, and is about to write the number $n + 1$ next. What is the sum of numbers already written?

Yunli would have wrote the number 1 once, 2 twice, and so on, until n n times. The sum of numbers should be $1 \cdot 1 + 2 \cdot 2 + \ldots = \frac{n(n+1)(2n+1)}{6}$, and there would be total $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ numbers written total.

This motivates us to find the largest n such that $\frac{n(n+1)(2n+1)}{6} < x$. We can use binary search for this purpose. The remaining numbers will be some copies of $n + 1$. Since we want the sum to be at least x, we take whatever remainder we have, and take the ceiling of that divided by $n + 1$ and add it to $\frac{n(n+1)}{2}$ to obtain our answer. The total runtime is $O(\log(X))$ per test case.

C. Leafy Distance

As per the statement, let's start a multi-source BFS starting from each leaf node. We can use a queue to process elements from closest to furthest to a leaf.

After the BFS is done, we can simply print the first occurrence of the largest number in our distance array. The time complexity is $O(n)$.

D. Counting Minimal Graphs

We can notice that for each node $i > 1$ that is distance d from node 1, there should be exactly one edge connecting it to a node that is distance $d - 1$ from node 1. This condition is obviously necessary (since there needs to be a way to get to this node from a node that is distance $d - 1$ away), and avoids any redundant edges.

Thus, the algorithm is as follows. For each node i that is distance d away from node 1, let x be the number of nodes that is distance $d-1$. Multiply our running total by x. Of course, this also means that if there is a node that is distance d, but no node that is distance $d-1$, the answer should be 0. All of this can be done with sorting and keeping a counter.

E. Partitioning

Let's solve the problem for a fixed k first. Intuitively, to minimize the sum of scores, all arrays should be approximately the same length (if the length of one array is 2 shorter than another, we can move one element from the longer array to the shorter one, and our sum of scores will necessarily decrease since all elements are positive), and the largest numbers should go before the smallest numbers. Indeed, this is true by the Rearrangement Inequality.

Thus, our algorithm to solve for a single k should be to sort the array in reverse order, then add a multiplier of $\lceil \frac{i}{k} \rceil$ to the (1-indexed) ith element. This algorithm naively runs in $O(n^2)$ which will pass the first subtask.

For full credit, let's solve for batches of the same multiplier for each k. For a fixed k, we want a multiplier of 1 for elements $1, 2, \ldots, k$, 2 for elements $k+1, k+2, \ldots, 2k$, and so on. The sum of elements can be found in $O(1)$ using prefix sum array. Using this method, the time to process k should be $O(\frac{n}{k})$ time, so the total runtime will be $O(\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{n})$ time.

At first glance, this time complexity does not seem to be any better than $O(n^2)$. But we notice that we can approximate $\frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{n}$ as $n \int_1^n \frac{1}{x} dx \approx n \ln(n)$. Thus, the problem is solved in $O(n \log n)$ time.

F. Modular Madness

The problem relies on the following claim:

Claim: $a \bmod c$ is either a, or at most $\frac{a}{2}$.

Proof:

1. If $a < c$, then $a \bmod c = a$. 2. If $c < \frac{a}{2}$, then $a \bmod c < c$, so $a \bmod c < \frac{a}{2}$ 3. If $a > c > \frac{a}{2}$, then $a \bmod c = a - c < a - \frac{a}{2}$.

This is relevant because for any number x, its value changes at most $\log x$ times! Now the task remains to filter out useless operations, and only perform operations that change the value. My approach is to push all querying numbers onto a priority queue and loop through the modulos one from left to right, continuously popping from the priority queue while the maximum number is greater than the modulo, updating its value, and pushing it back onto the priority queue.

Since each number is popped no more than $logX$ times, the total time complexity is $O(n + q \log X \log q)$.

G. A Counting Problem

Looping through all k^n arrays is of course, out of the question. Instead, let's count the contribution to the answer if $a_i = j$ for all pairs of (i, j).

What are the conditions for $a_i = j$ to be a prefix minimum? Clearly, all numbers before i must be at least as big as j, and there are no restrictions for any number after i. So, the contribution is $(k-j+1)^{i-1}k^{n-i}$. Summing this over all possible (i, j) yields an $O(nk \log MOD)$ solution, where $\log MOD$ factor comes from exponentiating and inversing.

To optimize this, note that when j is fixed, this becomes a geometric sequence with ratio $\frac{k}{k-j+1}$ as i varies. Thus, let's use the geometric sequence sum formula to solve this in $O(k \log MOD)$ time.

H. Maximizing Pairs

Let's first solve for each k separately. It is not too hard to find that for each i, if there is also a card with label $k-i$, then Bronya should take i and $k-i$ as a pair to gain 1 point. If we model the input array as instead a frequency array (where b_i is the number of occurrences of i in the original input), then we should add $\min(b_x, b_{k-x})$ over all $x < \frac{k}{2}$ (and add a special case for $x = \frac{k}{2}$ for even k, where we instead add $\lfloor \frac{b_x}{2} \rfloor$ to answer of k). This algorithm runs in $O(n^2)$ and passes the subtask.

To solve the full task, let's process all frequencies from least frequent to most frequent. Is there a way to handle all numbers with the same frequency f at the same time?

It turns out there is. Let S be the set of numbers with frequency f, and let T be the set of numbers with frequency $f+1$. Then for each $s \in S, t \in T$, we should add f to the answer of $s+t$. Additionally, for each $s_1, s_2 \in S$ where $s_1 \neq s_2$, we should add f to the answer of $s_1 + s_2$.

This may seem to not be helpful, since we are still dealing with potential $O(n^2)$ pairs to add each time. Fortunately, the structure of this allows us to model this polynomial multiplication. Model S as a polynomial $S(x)$ of length n with 1 if $b_i = f$, and 0 otherwise, and model T as a polynomial $T(x)$ of length n with 1 if $b_i > f$, and 0 otherwise. Counting the first part can be done simply by multiplying $S(x)$ and $T(x)$. For the second part, multiplying $S(x)$ by $S(x)$ almost works, but you will overcount where $s_1 = s_2$. To account for this, it turns out subtracting $S(x^2)$ will adjust to the correct coefficients. In summary, our current solution is to loop through frequencies, skipping all frequencies where no numbers are that frequency. For frequencies f with at least one number, let's model $S(x)$ and $T(x)$ as stated above, and add $f \cdot (S(x) \cdot T(x) + S(x)^2 - S(x^2))$ to the answers.

What is the time complexity of the above solution? As it turns out, since the sum of frequencies is equal to n, there are at most $\sqrt{n}$ different frequencies. Polynomial multiplication can be sped up using Fast Fourier Transform to be done in $O(n \log n)$, so the complexity is $O(n\sqrt{n} \log n)$. Unfortunately, even still, our solution still does not pass due to the high constant factor of Fast Fourier Transform, so we make one last improvement.

Let's casework on whether the number of numbers with a specific frequency f is above or below $(n)^{\frac{1}{3}}$ (think of this as the frequency table of frequency). When the frequency is above this, we can use the polynomial method as described above, and when the frequency is below this, we can use pure brute force. Using the same analysis as above, since the sum of frequency sum to n, there should be at most $n^{\frac{1}{3}}$ frequencies with at least one number. So, the number of brute forces we must do does not

exceed $n^{\frac{2}{3}}$, and the time complexity of polynomial multiplication we should do does not exceed $n^{\frac{1}{3}} \log n$. In practice, we should set n to be larger, to around 500, since the polynomial multiplication works much slower than brute force, even if asymptotically brute force is worse.