Problem A. Broken Keyboard

Sum, for all pairs of letters from the first and second word, the distance on the alphabet wheel: $\sum_{i=1}^{n} \min((a_i - b_i) \bmod 26, (b_i - a_i) \bmod 26)$.

Note that in C/C++ the built-in modulo function works incorrect for the negative integers, so you need to work on that (for example, consider using $26 + a_i - b_i$ and $26 + b_i - a_i$ instead of $a_i - b_i$ or $b_i - a_i$, respectively).

Problem B. Row of Columns

Read the heights in the array, then set max to 0 and go through array from the end to the beginning. Each time when the height of the current column is greater, than current max, the column is visible (special case: the n-th column is always visible, but it is processed automatically because the initial value of max is 0), so in this case update max with the height of the current column and increase the number of visible columns.

Problem C. Segments and Subsets

Notice that the structure of segments represents a tree, and also, for a fixed set, the optimal answer is x - (the sum of the lengths of segments into which nothing is nested). The tree can be built using a stack, sorting events of the form start of segment/end of segment. Then we iterate through the tree node (which is a segment) and calculate its contribution to the answer. This is $x \cdot 2^n - 1 - (r(v) - l(v)) \cdot 2^{n-sz(v)}$. Here $l(v), r(v)$ are the start and end of the segment, $sz(v)$ is the size of the subtree of node v. The idea is that the length of the segment of node v needs to be subtracted when it is a leaf.

Problem D. Sum of Characteristics

We will iterate over l from right to left and for each possible r maintain the minimum on the corresponding segment in some data structure. The key observation is as follows: let a_t be some element to the right of l, and together these elements give the value $max(a_i + t, a_t + j)$. Then it makes no sense to check the elements that are to the right of t and greater in value. But then we can maintain a decreasing sequence for the element a_l, the indices of which are equal to $i_0 = l < i_1 < ... < i_k < i_{k+1} = n + 1$, and the values $a(i_0) > a(i_1) > ... > a(i_k) > a(i_{k+1}) = 0$. Let $f(i,j) = max(a_i + j, a_j + i)$, $fp(i,j) = min(fp(i,j-1), f(i,j)$ for $j > i+1$, and $fp(i,i+1) = f(i,i+1)$. Consider the subsegment $i_s + 1 ... i_{s+1} - 1$. Notice that the values in our data structure only decrease, therefore, we should update some prefix of this subsegment with the new value, which is equal to $fp(l, i_s)$. Therefore, we can search for the first value on the segment that is less than the given one, update it on the segment, and find the sum on the segment, all of which can be done by a segment tree. The problem we have is that k can theoretically be large. However, now let's remember about the randomness of the elements. It is claimed that in this case, k will be small and in practice reaches only 33.

Problem E. Random Permutation

Let's denote pos_i as the current position of the number i. Let $l = 1, r = n$ be the current segment under consideration. Obviously, as long as $l \le pos_1$ and $r \ge pos_1$, we will write the number 1 into the sequence A. In other words, from one end, we must reach pos_1 and remove the one. Then, we will move to the segment $(l, pos_1 - 1)$ or $(pos_1 + 1, r)$ for some l, r. Let $ans(l,r)$ denote how many sequences A can be obtained if we have permutation elements numbered from l to r. By the way, it is clear that we cannot obtain two identical sequences if we make different transitions from some state (segments from different sides of the minimum contain different elements, and segments from one side contain a different number of elements). How to calculate $ans(l,r)$? Let $m = min(l,r)$ be the minimum on the segment (l,r), then $ans(l,r) = \sum_{i=l}^{pos_m} ans(i, pos_m - 1) + \sum_{j=pos_m}^{r} ans(pos_m + 1, j) - 1$. It seems that this is cubic dynamics, which can be optimized to quadratic. However, let's show that the number of states in a random permutation is $O(n \cdot log(n))$. Let's denote $left(i), right(i)$ as the nearest elements to the left and right, respectively, of the position i that are smaller than the element at position i. For each element i that is the maximum for its subsegment, states $(i+1, i+1)$, $(i+1, i+2)$, ..., $(i+1, i+right(i)-1)$ can be added, as well as similar states up to $left(i)$ on the left. In other words, we are interested in the sum $right(i) - left(i)$ for all i. To prove the estimate of this quantity, we will consider the elements in ascending order. Let's assume that we are considering the element x, and at all positions in the permutation where the elements are not greater than x, there is a 1, and at other positions there is a 0. We are interested in the distance from pos_x to the nearest one on the left and right. Since the permutation is random, we can assume that the elements are approximately equidistant from each other, and the distance is $O(n/x)$. And the sum over all such x is $O(nlogn)$, which is what needed to be proved. To make the solution pass the time limit, it is necessary to optimize it using partial sums. For each state, we will find the range of indices to sum the dynamic values in the corresponding partial sum vector, and the complexity will be $O(n \cdot log^2(n))$.

Problem F. Game

$dp(i)$ – the answer for the first player, if they are at position i, $f(j)$ – the answer if the second player is currently making a move from j. Notice that to calculate $dp(j)$, you can take the minimum over f on the segment $i + C ... i + r(i)$, however, f is not fully calculated up to these positions. This can be dealt with by either performing the operation $min =$ on the segment for f each time and taking the maximum on the segment over f, or by noticing that if we jump to the left of $i + C$, the second player will return us to the minimum on the segment $i, i + C$, then jump to $i + l(i)$ to give them fewer options.

Problem G. Permutation and Queries

1st observation: the answer does not exceed n. Indeed, neighboring elements in terms of index or value already give a result not greater than n. Let's denote up as the upper bound of the answer. Since $|i-j| \cdot |p_i - p_j| \leq up$, then one of the factors definitely does not exceed $\sqrt{up}$. Then we can find $f(P)$ as follows: we iterate over i and iterate over j in the range from $i - \sqrt{up}$ to $i + \sqrt{up}$. We have iterated over the indices, and we also need to iterate over the values in the range from $p(i)$. For updates, we will act similarly, iterating over the necessary indices and values from the range for elements at query positions a, b. We will maintain an array $cnt(i)$ - how many pairs of indices give the value i, then we need the minimum i with $cnt(i) > 0$. This could be maintained with a set. However, a set could worsen our complexity. Notice that we need to perform array cnt update queries very quickly, as we have many of them, and we can afford to answer the query slower - an estimate of $O(\sqrt{up})$ operations will not worsen the complexity. Then we can build a sqrt-decomposition on the array cnt. Complexity - $O(n\sqrt{n})$. UPD: the solution with a set on the cnt array also passes, as the update does not always access the set. The solution with a map for cnt gets TL.

Problem H. Make a Palindrome

If the length is even, then it is necessary to check that the strings at each even position match as sets. If it is odd, then we can obtain a palindrome by permuting at each even position.

Problem I. Good Subsegments

In order to calculate the number of k-good subsegments (denote this quantity as ans_k), we fix one of the edges (left or right) and say that exactly k elements are equal to each other from this edge. Denote this quantity as $calc_k$. Then $ans_k = calc_1 + calc_2 + ... + calc_k = ans_{k-1} + calc_k$, and we can calculate the array ans based on the array $calc$. Now, to calculate $calc$ for each value val, we write down in a separate vector all the lengths of segments of consecutive elements with this value. We go from left to right through each vector and calculate the contribution to the answer for this value. At the same time, we agree that when moving from left to right, we count subsegments such that at least k are equal to each other on the right, and at least k are equal to each other on the left. When moving from right to left, we count that at least k are equal on the left, and strictly more than k are equal on the right (we will describe only the calculation of the first option, the second will be carried out almost similarly). Subsegments consisting entirely of the same elements are counted separately. Let's say for the value val the current length of the segment of the same numbers is len, then we are interested in the sum over all len_{prev} encountered before, $max(len_{prev} - len + 1, 0)$. We will maintain the sum and the number of all encountered values, as well as store the array $cnt(i)$ - how many values less than or equal to k have been encountered. Then through these quantities, we can calculate everything, and also update them cumulatively in $O(n)$.

Problem J. Funny Numbers

Let n is funny, then $n = x + x + 1 + x + 2 = 3x + 3 = 3(x + 1)$, so n is divisible by 3. If we take any positive integer that is divisible by 3, then it can be represented as $3k$, where $k \geq 1$, but $3k = (k - 1) + (k - 1) + 1 + (k - 1) + 2$, all three integers are non-negative (because $k - 1 \geq 0$).

So all that you need is to check if n is divisible by 3.

Problem K. Maximize the Minimum

1. merge all into one array with indication of which array each element comes from. The minimum will be achieved between adjacent elements taken from different arrays.

2. binary search for the answer

3. dynamic programming on the array from the first step with additional flags and prefix optimizations in $O(n)$.

Problem L. Permutations and Cycles (Maximum Version)

Let's consider an identity permutation. We pair up the elements n and $n - 1$, as well as $n - 2$ and $n - 3$, $n - 4$ and $n - 5$, and so on. We call a pair bad if its sum is greater than x. Let the last k pairs be bad. Also, let a fixed point be an element that remains in its place, a movable one be an element that will be moved to another place, a heavy element be one of the rightmost $2 \cdot k$ elements, and a light element be one of the leftmost $n - 2 \cdot k$ elements. Let's consider and prove several statements:

- Two heavy elements cannot stand together - obvious (since the two smallest heavy elements form the leftmost bad pair).
- In each bad pair, there must be at least one movable point - follows from the first statement, and there must also be at least one light element.
- There are no fewer light elements than heavy ones.
- We can achieve the result in $n - k$ cycles: it is enough to take the left element of each bad pair s and swap it with the element $n - s$.

We will show that this is the optimal result and demonstrate the structure of the optimal permutation. Let's consider the following statements:

- In the optimal solution, the number of fixed points is not less than $n - 2 \cdot k$. This is true, because if there are no more than $n - 2 \cdot k - 1$ fixed points left, there are $2 \cdot k + 1$ elements remaining, from which a maximum of k cycles can be formed, i.e., there will be no more than $n - k - 1$ cycles, which is less than the found value.
- In the optimal solution, a heavy element is either a fixed point or stands in the position of a light element. Indeed, suppose we placed a heavy element w in the position of another heavy element. In the optimal solution, the total loss of fixed points should not exceed $2 \cdot k$. In each bad pair, there must be a light element, so the loss is already k. Therefore, the remaining loss should not exceed k. A heavy element must leave each bad pair, so an additional k is added to the total loss. However, from the pair where we place our heavy element w, two heavy elements must leave (and one light

element must come) - obviously, one must vacate the position for w, and the other cannot stand next to w. Therefore, the total loss becomes too large, and we cannot place w in the position of another element.

- In the optimal solution, a light element is either a fixed point or stands in the position of a heavy element. Similarly to the proof of the previous point, a light element that should stand in the position of another light element makes the loss too large, as it is already not less than $2 \cdot k$ due to each bad pair adding 2 to this loss.
- In the optimal solution, each bad pair contains a light and a heavy element. The proof is similar to the previous point.
- In the optimal solution, all cycles have a length of 1 or 2.

From the previous statements, it follows that all cycles have an even length, and the type of elements (light/heavy) alternates when traversing the cycle. Therefore, each cycle of length len results in a loss of len fixed points, i.e., the same as could be lost in the optimal solution. However, the total number of cycles decreases, so we end up with fewer cycles in the end.