Problem A. A Problems and A Balloons

If $k \geq n$, then each team can submit one problem — there are enough attempts. Thus, the answer is n. Otherwise, teams submit one problem each until the number of balloons runs out, meaning at least k teams will submit one problem. Therefore, the answer is the minimum of n and k.

Problem B. Big Alphanumeric String

It is obvious that the required number is composed of the maximum blocks of identical digits (otherwise, one could add a digit to the block and the number of "differing" pairs of neighbors would decrease). The blocks can be rearranged in $k!$ different ways, where k is the number of distinct digits in the number.

Problem C. Count The Sum Of GCD

The first observation we need is that the sum of the elements is divisible by the greatest common divisor. From this and the equality, it follows that X is divisible by g. Therefore, let's assume that we will iterate over g as a divisor of X (although this will not work in terms of time for very large X).

Dividing both sides of the equation by X, we isolate the numbers in the interval $[L, R]$ that are divisible by g, and divide them by g, so the problem reduces to selecting a subset of c elements from the new interval $[L_1, R_1]$ whose GCD is 1 and satisfies the equality $c + s = X_1$, where $X_1 = \frac{X}{g}$.

Let's increase all elements from the new interval by 1, obtaining a new interval $[L_2, R_2]$, where $L_2 = L_1 + 1$, $R_2 = R_1 + 1$. Now we need to select c elements from the interval $[L_2, R_2]$ such that their GCD is 1 and the sum $s = X_1$.

Let $low_c = L_2 + (L_2 + 1) + ... + (L_2 + c - 1)$ – the smallest possible sum that can be obtained, $high_c = (R_2 - c + 1) + (R_2 - c + 2) + ... + R_c$ – the largest possible sum. It can be noted that all intermediate sums are achievable, and also $low_c \leq low_{c+1}$ and $high_c \leq high_{c+1}$. Therefore, we can use binary search to find the range that contains X_1, and add all such c to the answer.

An unresolved problem so far is that the subset may not be coprime.

The key statement is that for $c \geq 3$, if there exists a subset of c elements with sum s, then there also exists a subset with such c and s and with GCD equal to 1.

Let's prove this. Suppose the subset contains both even and odd elements. Take two adjacent even and odd elements and move them 1 unit closer to each other. The sum will not change. We will keep moving them until their difference equals 1. Now the GCD is 1.

Suppose there are only odd elements. Take two adjacent odd elements and move them towards each other as well, after which they will either be 2 units apart or 4 units apart. The GCD will thus be at most 4. Since the elements are odd, it cannot be 2 or 4. It also cannot be 3, since the considered pair of odd elements has different remainders when divided by 3.

Suppose we have only even elements. If there is a pair of elements at least 4 units apart, we can move them towards each other by 1 and reduce it to the previous cases, as they will now be odd. If all even elements are 2 units apart, then we can take any three consecutive elements and move the first and third towards each other. We have proven it.

If $c = 2$, we can move the elements 1 unit towards each other. We cannot do this if the first element is 1, but in this case, the GCD is already 1, or if the last element is R_1. We find the only case where a subset of two elements with sum s cannot be coprime is $s = R_1 + (R_1 - 2)$.

Now let's return to how to iterate over g. Notice that $g \cdot c \leq R$. Therefore, if $g^2 \leq R$, we can iterate over g up to the square root, otherwise c up to the square root. Let's consider the second case in more detail.

If $c \leq \sqrt{R}$, then we note that $s \leq R \cdot c \leq R\sqrt{R}$.

We then obtain that $X \leq R + R\sqrt{R}$. Therefore, if $X > R + R\sqrt{R}$, then c cannot be less than the square root, and there is only one case that needs to be considered here – $g \leq \sqrt{R}$. Otherwise, X is not very large, and we can iterate over the divisor of X.

Problem D. Death Star N

Four points are in the same plane if and only if the volume of the tetrahedron formed by them is zero. As is known, the volume of a tetrahedron is calculated as the determinant of a 3×3 matrix constructed from vectors emanating from one of the vertices of the tetrahedron (in our case, it is natural to take the vertex (0,0,0)).

The formula for the determinant of a 3×3 matrix is:

$$a_{00}a_{11}a_{22} - a_{00}a_{12}a_{21} - a_{01}a_{10}a_{22} + a_{01}a_{12}a_{20} + a_{02}a_{10}a_{21} - a_{02}a_{21}a_{31}$$

We will show that if at some point, when adding numbers between 0 and 1 (excluding 1), the determinant becomes 0, then adding 1 to the corresponding variables will change its sign.

Indeed, it can be noted that for each of the variables, the determinant is a linear function, meaning that if it changes in a certain direction when changing from 0 to $0 < x < 1$, it will continue to change in the same direction when changing from x to 1, so the sign of the determinant of the original matrix will change when 1 is added instead of x.

From this, it also follows that if adding x_{ij} to some variables changes the sign, then the sign will remain changed when adding 1 to those variables.

Conversely, if adding 1 to some set of variables changes the sign to the opposite, we can replace the 1 in the expression with x (considering that we are appending the same "tail" to each variable) and obtain a polynomial function of one variable. Since this function has different signs at $x = 0$ and $x = 1$, it must have at least one root in $(0, 1)$.

Thus, we first calculate the initial determinant. If it equals zero, then we can change nothing, and the answer will be 0 1. Otherwise, we assign one bit to each variable and iterate through the numbers from 1 to 511, sorted by the number of bits in their binary representation (the value of the function `__builtin_popcount` in C++). If at some point the sign of the determinant changes to the opposite (note that if it becomes zero, that is not enough, as we need to add numbers strictly less than one), then this set of variables allows us to "zero out"the determinant. We continue iterating while the number of variables is equal to the number of variables in the first found set and find the number of ways. If the sign never becomes opposite, we output -1.

Problem E. Exclusions

Let's iterate m from 0 to n and count the minimum number of blocks into which the array can be partitioned if all mex values are no greater than m. Let this number be b, and the array of answers be ans. Then we need to update the prefix of the array ans up to b with the value m.

The number b can be found greedily, starting from the beginning each time, selecting the longest block where $mex \leq m$. Let to_i be the leftmost position such that the mex on the segment (i, to_i) is greater than m. We can recalculate to_i using a segment tree as m increases. Specifically, for to_i to change for some i, the old mex must equal the new m. We will go through all positions of m in the array. The occurrences of m divide the array into groups of consecutive elements that do not contain m. Notice that there is monotonicity at to_i, so for each group, we will find out to which position we need to update the prefix for that group by descending through the segment tree.

In total, we will make $n \cdot \log n$ jumps across all m due to the estimate of the partial sum of the harmonic series. Additionally, we access the segment tree in logarithmic time, resulting in a complexity of $n \cdot \log^2 n$.

Problem F. Fix The Leak!

- Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.
- Idea: Removing a leaf only reduces the count if it has siblings.

- Greedy algorithm: Repeatedly remove the shortest leaf-branch.
- Idea: Under each vertex, the deepest path is always removed last.
- Solution: Using DFS or bottom-up dynamic programming, find the length of the deepest path under each vertex. At each vertex, increase the length of the deepest child node by one and mark the paths of other child nodes as final. Sort the final lengths and count how many of them sum up to not exceed k.
- Time complexity: $\mathcal{O}(n)$.

Problem G. Governor Scroogius

To obtain a partition with the maximum sum, it is sufficient to simply break the string into blocks of one digit. The reduction in sum can only be achieved through "inversions" like IV, XL, and so on. Each inversion takes at least two digits. Depending on the first digit in such a notation, an inversion can yield a "gain" of 2, 20, or 200. Thus, if n is odd, there is no solution.

If n is even, then $n/2$ must be represented as a sum of the smallest number of 1s (represented by the pair `IV`, since $V < X$ lexicographically), 10s (represented by the pair `XC`, since $C < L$ lexicographically), and 100s (represented by the pair `CD`, since $D < M$ lexicographically). Since if we have more than 10 units, they can be replaced by one 10, and if we have more than 10 tens, they can be replaced by one hundred, the number of pairs `IV` will be $n/2 \bmod 10$, the number of pairs `XC` will be $[n/20] \bmod 10$, and the number of pairs `CD` will be $[n/200]$. The remaining task is to arrange them in lexicographical order, that is, first all `CD`, then all `IV`, and finally all `XC`.

Problem H. Headquarter Reforms

Formalization of Auxiliary Functions and Data Structures

Let us introduce an auxiliary function $f'(l, i)$ as the largest integer t satisfying the conditions:

- $l \geq i - t + 1$
- All elements of the sequence S from the $(i - t + 1)$-th to the i-th are equal to 0.

Define the auxiliary function $g'(l, i)$ as follows:

$$g'(l, i) = (A_i + B_i \times f'(l, i)) \times (1 - S_i)$$

Structure of the Segment Tree Node with Lazy Operations

Each node of the segment tree managing the interval $[l, r)$ will store the following data:

1. $\sum_{i=l}^{r-1} g(l, i)$
2. $\sum_{i=l}^{r-1} g'(l, i)$
3. The number of indices i $(l \leq i \leq r - 1)$ for which the condition $S_l = S_{l+1} = \cdots = S_i = 1$ holds
4. The number of indices i $(l \leq i \leq r - 1)$ for which the condition $S_l = S_{l+1} = \cdots = S_i = 0$ holds
5. The sum of values B_i for all indices i $(l \leq i \leq r-1)$ for which the condition $S_l = S_{l+1} = \cdots = S_i = 1$ holds

6. The sum of values B_i for all indices i ($l \le i \le r-1$) for which the condition $S_l = S_{l+1} = \cdots = S_i = 0$ holds
7. The number of indices i ($l \le i \le r-1$) for which the condition $S_i = S_{i+1} = \cdots = S_{r-1} = 1$ holds
8. The number of indices i ($l \le i \le r-1$) for which the condition $S_i = S_{i+1} = \cdots = S_{r-1} = 0$ holds
9. The sum of values B_i for all indices i ($l \le i \le r-1$) for which the condition $S_i = S_{i+1} = \cdots = S_{r-1} = 1$ holds
10. The sum of values B_i for all indices i ($l \le i \le r-1$) for which the condition $S_i = S_{i+1} = \cdots = S_{r-1} = 0$ holds

This data is sufficient to process queries of type $c = 2$.

Processing Queries of Type $c = 1$

For a query with $c = 1$ (inverting values of S over the interval $[l, r]$), it is necessary to swap the values in the corresponding nodes of the segment tree related to the conditions for ones and zeros (i.e., the data in points 3 and 4, 5 and 6, 7 and 8, 9 and 10).

Problem I. Infinite Game

Firstly, since the maximum number of "forbidden"digits is two (one repeats the last digit, the other makes the sum of the digits divisible by three), and we have three digits, a move can always be made, meaning the game is infinite.

Thus, we need to determine the number of possible positions at the n-th move. Note that the state of the game is defined by two parameters — the last digit and the remainder of the sum modulo 3. Therefore, there are a total of 9 states. We can analyze them directly:

- From the state (0,0) you can transition to either state (2,2) or state (1,1). Appending a zero will violate both rules.
- From the state (0,1) you can only transition to state (2,2): appending a zero will give divisibility by 3, appending a one will cause a repeat.
- From the state (0,2) you can only transition to state (1,1): appending a zero will give divisibility by 3, appending a two will cause a repeat.
- From the state (1,0) you can only transition to state (2,1): 0 — repeat, 2 — divisibility.
- From the state (1,1) you can only transition to state (1,0): 1 — repeat, 2 — divisibility.
- From the state (1,2) you can transition to either state (2,1) or state (1,0): 1 — both repeat and divisibility.
- From the state (2,0) you can only transition to state (1,2): 0 — repeat, 1 — divisibility.
- From the state (2,1) you can transition to either state (1,2) or state (2,0): 2 — both repeat and divisibility.
- From the state (2,2) you can only transition to state (2,0): 2 — repeat, 0 — divisibility.

Next, the problem can be solved using dynamic programming.

However, there is also a more mathematical approach: it can be noted that after some non-periodic prefix (of length no more than three operations), the written sequence begins to take the form 121212... or

212121..., where between each pair of digits there may or may not be exactly one zero. Since the non-zero digit is uniquely defined, this is equivalent to the problem of "counting the number of sequences that do not contain two zeros next to each other"; as is known, this is the Fibonacci sequence. Therefore, the result is equal to the $(n - f)$-th element of the Fibonacci sequence (where f is the length of the fixed part of the sequence before the start of the regular part), except for the case (0,0), where the answer is doubled (since at (0,0) both variants of regular chains are possible).

Problem J. Just One Attempt...

By calculating all the sums of the digits for Fibonacci numbers up to 2^{61} (there are 88 in total), we note that the pair of sums of digits for two consecutive numbers is unique. We construct a map $sum_1, sum_2 \to fib_1, fib_2$.

Thus, if the sum of the digits does not equal the sum of the digits of the last of the numbers, we find the sums of the digits for `fib[x]` and `fib[x+1]` and recover the original number from this pair. Otherwise, we find the sums of the digits for `fib[x-1]` and `fib[x]` and recover the original number from this pair.

Problem K. Kenobi and Skywalker

Let a certain subset of c elements with xor equal to x be fixed. Let some value v occur in it t times. Then, if we remove $2, 4, 6...$ occurrences of v from the set, its xor will not change. Let's count cnt_i – how many times the value i appears in the array. Let u be the number of unique values in the subset. We can write a dynamic programming function of the following form:

$dp(val, xor, unique)$ – the maximum number of elements in a subset where all values are no greater than val, their xor is equal to xor, and there are $unique$ unique values.

Transitions:

$dp(val, xor, unique) = \max(dp(val-1, xor, unique)+even(cnt_{val}), dp(val-1, xor \oplus val, unique-1)+odd(cnt_{val}))$.

Here, $even(x)$ is the largest even number not exceeding x; $odd(x)$ is the largest odd number not exceeding x. The meaning of the transitions is that we can add another $2, 4, 6...$ occurrences of each value to the set without changing the xor.

After we compute such dynamic programming, let's allocate segments $[unique, dp(...)]$ of the counts of elements in the subset for each value of xor, where such xor is achieved. We need to consider even and odd counts separately and write a union of segments. After that, the answer to each query can be found using binary search.

Let $M = 511$. The complexity is $O(M^3 + n + q \cdot \log M)$.

Problem L. Limit The Arrow

If the thickness of the targets increases monotonically, a queue q can be used to store the thickness values of the targets and a variable s for the sum of the values in the queue. Targets are added to the queue sequentially, starting from the first. As t increases, the values in the queue increase monotonically. If after removing the first element from the queue the sum s remains at least p, then that element is removed. This process is repeated as long as possible.

For each i, we maintain the minimum size of the queue. If $S < P$, then $f(i) = -1$, otherwise $f(i) = |Q|$ (where $|Q|$ is the number of elements in the queue). Time complexity: $O(N)$.

In the general case (if the thickness of the targets is distributed arbitrarily), we use a heap instead of a regular queue to work with arbitrary input order. Time complexity: $O(N \log N)$.

Problem M. Minimum Number of Tasks

Since the task letters are consecutive, there cannot be fewer tasks than the task with the highest number among those that the participant attempted to submit. We know nothing about the other tasks, and

to minimize the total number of tasks, we can simply assume that those tasks do not exist. Thus, it is sufficient to find the letter with the highest number (i.e., with the highest code) and output that number. The easiest way to do this is to keep track of the maximum ASCII code of the letter and subtract the code of the letter `A` minus one when outputting.