Problem A. Algebraically Universal Numbers

Let $a^x = b$ and $a^y = b$ with $x > y$ (without loss of generality). Then $a^y(a^{x-y} - 1) = 0$, which means either $a = 0$, or $a = 1$, or $a = -1$ (in which case $x - y$ must be even). Thus, the only algebraically universal numbers are 0, 1, and -1.

Problem B. Conditions

Let's learn how to find the next integer that might be a fitting one. If the current value has 7, then go to the next value without 7 or the next value divisible by k, whichever is smaller. In the second case, while the current number is in the list and has 7, keep adding k. In both cases, as you reach a value without 7, keep checking it and the next values until you reach a value with 7 again.

Problem C. Modulo 4

We consider expressions as lists of values. The bits in the values are numbered from least significant to most significant, starting from 1.

A bit in a certain context is called free if its value does not affect whether the expression belongs to C_n. There are three reasons why a bit can be free:

- it is the $n + 1$-th or higher bit;
- to the right of the value with this bit, there is a value with more than n bits;
- from the context, it is known that in another value, the bit at the same position is set to 1.

The most significant bit cannot be free.

Let L be the length of the longest value in the expression.

The number of expressions with $L > n + 2$ is 0, as each such expression has at least two free bits.

If $L = n + 2$ and there is at least one value of length greater than 1 to the left of the long value, then there are two free bits in this expression. If $L = n + 2$ and there is a value of any length $j \le n$ to the right of the long value, then in the long value, the bits $n + 1$ and j are free. Thus, we are only interested in two expressions with $L = n + 2$: `1|1|...1|1011..1` and `1|1|...1|1111..1`, adding 2 to the answer.

Let's consider expressions with $L = n$, and later we will return to the case $L = n + 1$. If the expression contains two values of lengths i and j ($i < j < n$), then it also contains a value of exactly length n in which the bits i and j are free. If the expression contains a value of length $j < n$ and two values of length n, then in both, the bit j is free. Thus, we are interested in two types of expressions: consisting only of values of length n, and consisting of one value of length n and several values of length $j < n$.

Expressions of the first type exist only if $n|k$. In $\frac{k}{n}$ values, the bits from the first to the second-to-last can be set in $2^{\frac{k}{n}} - 1$ ways, thus the number of expressions of this type is $(2^{\frac{k}{n}} - 1)^{n-1}$. This number is equal to 0 for $k = 0$, 1 for $k = n$ or odd n, and 3 otherwise.

For each j from 1 to $n - 1$, expressions of the second type exist only if $j|(k - n)$ and $k > n$. In a value of length n, the j-th bit is free. In $1 + \frac{k-n}{j}$ values, the bits from the first to the $k - 1$-th can be set in $2^{1+\frac{k-n}{j}} - 1$ ways, and the long value can be placed $1 + \frac{k-n}{j}$ ways between the short ones. Multiplying 2, $(2^{1+\frac{k-n}{j}} - 1)^{k-1}$ and $1 + \frac{k-n}{j}$, we get 2 if $k - n$ is divisible by $2j$, otherwise 0. Thus, the number of expressions of the second type is 0 if $k - n$ is odd, otherwise, it is twice the number of divisors of $\frac{k-n}{2}$ that are less than n, which can be calculated in $O(\sqrt{k-n})$. Let's denote the total number of expressions of the first and second types as $t(k, n)$ and add this to the answer.

Let's return to the case $L = n + 1$. Let i be the position at which the long value starts, then the number of such expressions is equal to $\sum_{i=1}^{k-n} f(i-1)g(k-i, n)$, where $f(x)$ is the number of expressions of length x with at most one free bit if all bits below the most significant are considered free; $g(x, n)$ is the number

of expressions of length $x+1$, where the first value has length $n+1$, and the rest are no longer than n. Let's consider $g(x,n)$. The case $x=n$ is trivial, $g(n,n)=1$, let's study the cases $x>n$.

If the first value looks like `100...00`, then the number of such expressions is equal to $t(x-n,n)$.

If the first value has the n-th bit, then similarly to the calculation of $t(k,n)$, we add $(2^{\frac{x}{n}}-1)^{n-1}$ if $x=0 \bmod n$, and 2 for each divisor of the number $x-n$ (not $\frac{x-n}{2}$, because the long value has only one way to be placed) that is less than n.

If the first value has the j-th bit, but not the bits from $j+1$ to n, then similarly to the calculation of $t(k,n)$, we add 2 if $\frac{x-2n}{2}$ is divisible by j; in this case, $x-2n=0$ is allowed; in other words, we add 2 for each divisor of $\frac{x-2n}{2}$ that is less than n.

We get that $g(x,n)$ consists of the following summands:

- $(2^{\frac{x-n}{n}}-1)^{n-1}$ if $n|x$;
- $2(n-1)$ if $x=2n$;
- $(2^{\frac{x}{n}}-1)^{n-1}$ if $n|x$;
- $2\cdot\#\{d\in\mathbb{N}|0<d<n\wedge d|(x-n)\}$;

It is also possible to add the first and third terms:

$$(2^{\frac{x-n}{n}}-1)^{n-1}+(2^{\frac{x}{n}}-1)^{n-1}=\begin{cases}1^{n-1}+3^{n-1}=2\cdot(n\%2), \text{ if } x=2n\\ 3^{n-1}+3^{n-1}=2, \text{ if } x>2n\wedge n|x\end{cases}$$

Calculating $f(x)$ is not difficult: either the expression consists of x ones, or of $x-2$ ones and `10` or `11`, which can be placed in $x-1$ ways. Thus, $f(x)=\max(2x-1,1)$, which is 1 for $x=0$ and odd x, and 3 otherwise. It can also be noticed that $g(k-i,n)$ is even if $k-i>n$, so for $i<k-n$, it doesn't matter whether $f(i-1)=1$ or $f(i-1)=3$.

Let's consider how to calculate $\sum_{x=1}^{k-n-1}g(k-x,n)=\sum_{x=n+1}^{k-1}g(x,n)$ (the term equal to $f(k-n-1)\cdot g(n,n)=f(k-n-1)$ should be calculated separately). Simplifying the first and third terms, it is easy to sum them for all x.

Let's calculate $\sum_{x=n+1}^{k-1}2\cdot\#\{d\in\mathbb{N}|0<d<n\wedge d|(x-n)\}$:

$$\sum_{x=n+1}^{k-1}2\cdot\#\{d\in\mathbb{N}|0<d<n\wedge d|(x-n)\}=2\cdot\#\{(d\in[1;n),x\in[n+1;k-1])|x\equiv n \bmod d\}=2\sum_{d=1}^{n-1}\left\lfloor\frac{k-n-1}{d}\right\rfloor$$

This sum is calculated by a known algorithm in $O(\sqrt{k-n})$. To get the final answer, we need to sum the terms calculated at different stages of the solution.

Problem D. Blind Gauss

How to find a matrix with an odd determinant:

It's equivalent to finding a *nondegenerate* matrix in $\mathbb{Z}_2$ with the required number of 1s in each row. Btw, at this point, it's obvious for which cases the answer is -1, when there are at least two a_i equal to n, or when all the a_i are even.

Assuming the answer exists, generate each row randomly until it's linearly independent of previously generated rows. Oh, and if you have $a_i=n$ in the input, then generate that row before others.

How to get from an odd determinant to 1:

You have a matrix of 0 and 1 with the required number of 1s in each row. Run Gaussian elimination (in $\mathbb{Z}_2$) to get a triangular matrix.

Go back to $\mathbb{Z}$ and run the steps of the performed elimination in the reversed order. You'll get a matrix with the required number of odd numbers and its determinant is 1 or -1. In the latter case, just swap any two columns.

Problem E. Alice, Rooks, and Pawns

To calculate the number of ways Alice can place k pawns on the board, we first need to determine the number of safe cells. Considering memory constraints, we will use two bit arrays to mark the rows and columns that are under attack by rooks.

The problem reduces to the following: *In how many ways can we place k pawns in x safe cells?* The number of ways is equal to the number of combinations with repetitions:

$$\binom{K+X-1}{X-1}$$

To solve the task, we will apply modular division and fast exponentiation.

Problem F. Grand Prix of Array Count

Obviouisly, all the values $a_2, a_3, \ldots, a_n$ should be equal to each other.

If n is odd, then selecting two values a_1 and a_n forces us to set a_{middle} to their gcd, thus the answer is k^2.

If n is even, then after selecting a_{middle} we have k/a_{middle} options to choose a_1 and the same amount of options to choose a_n, thus the answer is $\sum_{i=1}^{k} \left(\frac{k}{i}\right)^2$. It can be calculated in $O(\sqrt{k})$ with the famous algorithm.

Problem G. Fans

0.1 Subtask 1

We use the constraint that the number of fans who reach the stadium is up to 10^8. This means we do not need to consider cells at very large distances. To ensure this, we explore all cells with coordinates whose absolute values are ≤ 2500 (a total of $5000 \times 5000 = 2.5 \times 10^8$ cells).

One approach is to run a *Breadth-First Search (BFS)* from each cell to the center. Although this is not optimal here, we can optimize by running a single *BFS* from the center outward, since the grid (with cells as vertices and edges representing possible fan movements) is undirected.

We use a 5000×5000 matrix for the cells we traverse, adding 2500 to the coordinates to avoid negative numbers. For each cell, we check the parity of the path length to determine the fan's allegiance. The complexity of this algorithm is $O(\text{ans})$, where ans is the number of fans arriving at the gathering.

0.2 Subtask 2

Given that obstacles have coordinates with absolute values ≤ 1000, we refine our approach:

- We limit the *BFS* to cells with coordinates ≤ 1001 in absolute value. This covers obstacles in the square and finds optimal paths from each cell in this square to the center.
- For cells outside this square, we determine their optimal paths by connecting them to the nearest cells on the boundary of the square. This is based on the observation that any optimal path from an external cell must eventually enter the square through the corresponding boundary cell.
- For corner regions, we connect external cells to the nearest corner of the square. The parity of distances alternates as we move outward, allowing us to efficiently count even and odd distances.

0.3 Subtask 3

Note that the algorithm from subtask 2 has complexity $O(N+S)$, which is too slow for the full solution. Further optimization involves deriving a formula for counting even/odd cells, noting that the number of such cells decreases by 1 every two rows. This reduces the complexity of the final solution to $O(N)$.

Problem H. Spruce and Oak Tiles

Let $A_{i,j}$ denote the maximum possible length when considering only the top and left borders with cell (i,j) as the top-left corner, and $B_{i,j}$ denote the maximum possible length when considering only the bottom and right borders with cell (i,j) as the bottom-right corner.

Both $A_{i,j}$ and $B_{i,j}$ can be computed using techniques such as two pointers or binary search.

To determine the maximum frame length using $A_{i,j}$ and $B_{i,j}$, we consider each value of $i+j$ independently and find the maximum among them. The solution involves computing the maximum value for each $i+j=k$.

For all (i,j) satisfying $i+j=k$, let A' be the sequence of $A_{i,j}$ sorted in increasing order of i, and B' be the sequence of $B_{i,j}$ sorted in increasing order of i.

A frame of length $q-p+1$ can be constructed if there exist indices p,q such that:

- $A'_p \geq q-p+1$
- $B'_q \geq q-p+1$

This can be achieved by maintaining a set of values A'_j where $j+A'_j \geq i$, and for each i finding the minimum value in the set that is greater than or equal to $i-B_i$.

If a balanced search tree is used to represent the set, this procedure can be implemented for all k with a total time complexity of $O(HW\log(H+W))$.

- Each of the $H\times W$ elements is inserted into and removed from the set at most once.
- The maximum number of pairs (i,j) satisfying $i+j=k$ is $H+W-1$, so the maximum number of elements in the set is $O(H+W)$.

Problem I. Try This at Home

Compress the values to the range $[0;base)$, where *base* is the amount of different values in the array.

Obviously, as long as array a contains each value at least twice, then applying function f is equivalent to adding 1 in base *base*. Thus, we will find the next array that doesn't contain each initial value at least twice, and we will subract one array from another in base *base*. The only exception is when all the maximum values are located in the tail of the array, but in this case, applying f to it immediately gives us the required result.

To find the next array that doesn't contain each initial value at least twice, find the latest second entry among all elements, let it be at the index *pos*. Obviously, it has to be changed, so set all the values after *pos* to *base* -1 and add 1 in base *base*. Repeat until you get the required array. You have to think of all the corner cases to be able to repeat this operation in O(1).

Problem J. Pizza Restaurant

1) Case $|s_x| \geq k\cdot|x_y|$: In that case, $s_x = k\cdot$ reversed s_y + *some palindrome*. So, for each string in the input as s_x, for each its prefix, if the reversed prefix exists in the input, try to concatenate that prefix with itself as long as the concatenation keeps being s_x prefix, and at each step, check that the remaining suffix is a palindrome.

2) Case $|s_x| < k\cdot|s_y|$: There are two subcases:

- there is only one acceptable k, and it's equal to $ceil(|s_x|/|s_y|)$, (for example, *abccab ccba ccba*;
- each integer k starting from 1 is acceptable

We will check if it's possible with $k = 1$, as it covers both the second subcase and the first one if $|s_x| < |s_y|$. If $|s_x| \geq |s_y|$, then $k = 1$ was already covered by the first case. If $|s_x| < |s_y|$, then iterate over input strings as s_y, for each its prefix, if it's a palindrome and the reversed remaining suffix is in the input, we found it.

The only thing left is to check the subcase $|s_x| > |s_y|$ and $k = ceil(|s_x|/|s_y|)$. For each string in the input as s_x, for each its prefix, if the reversed prefix exists in the input, check if s + *reversed prefix* k *times* is a palindrome (use hashes), where $k = ceil(|s|/|prefix|)$.

Problem K. Unified Excursion Ticket

First, we observe that we will always visit some segment containing the starting position x. If the visited subsegment is $[L, R]$, then the final bitwise AND will be A_L AND A_{L+1} AND ... AND A_R (denoted as $f(L, R)$), and the minimum number of steps to visit it is $R - L + \min(x - L, R - x)$ (going from the start to the left boundary, then to the right, or vice versa).

Enumerating all possible left and right boundaries, computing the AND, and traversing all suborders, we get a solution with complexity $\mathcal{O}(QN^3)$.

If we maintain the values $f(L, x)$ and $f(x, R)$ and update them during iterations, then we do not need to traverse the entire segment to find $f(L, R)$, as we can compute it from these saved values, improving the estimate to $\mathcal{O}(QN^2)$.

Next, note that the optimal subsegment will always have its boundaries at positions where the AND value on the segment changes, i.e., at L and R such that $f(L, x) \neq f(L + 1, x)$ (or $L = x$) and $f(x, R) \neq f(x, R - 1)$ (or $R = x$). This is true because if the boundary position does not change the AND, we can simply exclude it from the subsegment and get a smaller subsegment.

Positions that change the AND value are those where some bit is missing in its element but present in all elements from it to x.

For each bit, we can find the nearest left and right positions whose elements do not contain this bit in one pass through the array (denote them as l_i and r_i for the i-th bit). Our interval must contain at least one of these. We can formulate it as: if the left boundary is greater than l_i, then the right boundary must be at least r_i.

We iterate through the sorted list of potential left boundaries (of which there are only $\log(A_i)$), maintaining the current minimum right boundary, thus finding the minimal solution.

Finally, to find the positions of zeros faster, for each bit we maintain a set of positions whose elements do not contain this bit.

Overall complexity: $\mathcal{O}(Q \log(N) \log(A_i))$