Problem A. Hard to Compare

We will start with the obvious formula:

$$f(n,k,x) = \frac{n!(k-x)!g(n-x,k-x)}{(n-x)!},$$

where $g(n-x,k-x)$ represents the amount of arrays of length $n-x$ containing $k-x$ different values at least twice each, up to renumerating the values.

The integer values of the function g can be calculated with this dp:

$$g(x,y) = g(x-1,y)\cdot y + g(x-2,y-1)\cdot(x-1).$$

As the memory limit is small, only keep every 25-th diagonal; this will take 4 MB. We keep the diagonals because if we know a diagonal, we can restore the next one; moreover, all the pairs $(n-x,k-x)$ lie on the same diagonal for fixed n,k.

To find the largest values, we will compare their logarithms. The function $ln\ f(n,k,x)$ is concave for any fixed n,k (it's unobvious, but it's true!). Thus, to find the 9 greatest values, do the binary search to find the smallest x such that $ln\ f(n,k,x) > ln\ f(n,k,x+9)$. The values $f(n,k,x), f(n,k,1), \ldots, f(n,k,x+8)$ are the greatest ones. If such x does not exist, then the function is an increasing one, so the values $f(n,k,k-9), f(n,k,k-8), \ldots, f(n,k,k-1)$ are the biggest ones.

The logarithm $ln\ g(n-x,k-x)$ should be calculated in $O(1)$ time and memory. To achieve that, precalculate all the logarithms locally, and run your favourite machine learning algorithm to get a regression that approximates the function $g(x,y)$ very precisely. In the author's solution, the space of arguments is divided in 58 sectors; for each of these sectors, a third degree polynomial regression is applied with the features $x, y, ln(x), ln(y), ln(x-2y+1)$, and some $L1$ regularization. At each sector, the mean absolute error $\le 10^{-5}$ has been achieved; thus, the approximation is precise enough to compare any two values $f(n,k,x)$ and $f(n,k,x+9)$. Participants can validate all the comparisons locally and fix the wrong outcomes (if there are not many of them) just by hardcoding the actual values. The author's solution required to fix two such values.

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 $x = 0 \mod n$, 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. Eternal Masters

Let's assume $\sum r_i < L + \sum w_i$, otherwise Red easily wins by dumping all their cards. Now, let's assume Red is able to win. Let's observe the final game state immediately after Red's victory. Obviously,

- there are White's cards in the stack,
- there is the same amount of Red's cards in the stack,
- the total value of White's cards in the stack is greater than the total value of Red's cards in the stack.

Moreover,

$$\sum_{r_i \notin stack} r_i - \sum_{w_i \notin stack} w_i \geq L_{initial}.$$

Note the current stack content, and let's go back in time to every moment when exactly that content had appeared. At each of those times, especially at the first one, the same inequality did hold, and Red could easily win by dumping all their cards. Thus, this inequality is a necessary and sufficient condition for Red to be able to win.

Let's denote

$$\sum_{r_i \notin stack} r_i - \sum_{w_i \notin stack} w_i$$

as D. Red has to play towards maximizing D, and White has to play towards minimizing it. However, for both players, any action leads towards the opposite. If Red plays a card r_i, then D is decreased by r_i; if

Red passes, then D is decreased by the value of the top White's card on the stack. The same is true for White, but their actions cause an increase instead of a decrease.

Summing up, the optimal game goes as follows:

1. When the stack is empty, Red plays their smallest card. (or loses if they don't have one).
2. If White has a card with a smaller value than the top Red's card on the stack, they play it, and Red passes. goto 2.
3. Else if White is able to pass, White passes. goto 1.
4. Else if White has no cards, White loses.
5. Else, White has to play a card with a value greater than the top Red's card on the stack. At this point, we can forget these two cards exist, as for Red it's never profitable to pass with these cards on top. goto 1.

This can be easily simulated to select the winner. Also, it's easy to adapt this strategy in case the interactor plays unoptimally.

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. 1 :eye: > 100 :ear:

First, we will discuss the algorithm to solve for any two nonconvex polygons in general, then we will talk about optimizing it for our input.

Rotate the input by a random angle to get rid of vertical edges.

The Minkowski sum of two polygons is the same as the Minkowski sum of their boundaries (that's not always true, but in this problem it is true due to the "special" generation). Thus the required area is the area of the union of nm parallelograms. These parallelograms can intersect in $(nm)^2$ points. Let $f(x)$ be the intersection of that union with the vertical line at x. Thus the area of the union is $\int_{-\infty}^{\infty} f(x)\,dx$. $f(x)$ is a piecewise linear function and it changes its first derivative at most $(nm)^2$ times. $f(x)$ can be calculated in $O(nm\ log\ nm)$ by intersecting that vertical line with every parallelogram and sorting the intersection points. This way we can also find the derivatives. So the area of the union can be found in $O(n^3m^3\ log\ nm)$.

For our input, we make a few assumptions:

- Since the initial points generation picks the points uniformly, in both 1000-gons, there are enough points near the edges of the $[0; 10^5] \times [0; 10^5]$ square. Therefore, the union of parallelograms looks like a square $[0; 2 \cdot 10^5] \times [0; 2 \cdot 10^5]$ with "broken" edges, and its area is somewhere between $3.80e10$ and $3.99e10$;
- In practice, the square $[20000; 180000] \times [20000; 180000]$ is fully covered with the parallelograms; we can remove all the parallelograms fully covered by this square, thus reducing the amount of parallelograms from 10^6 to somewhere between 40000 and 60000.

- In practice, there are around five to six hundred x coordinates where $f(x)$ changes its first derivative. These vertices can be detected with this combination of binary search and Newton's method:

We are calculating $\int_L^R f(x)\,dx$ for some L, R.

If $f'(L) = f'(R)$ and $f(R) = f(L) + f'(L) \cdot (R - L)$, then we add the area of that trapezoid between L and R to the answer. Well, there can be some outliers between L and R with different derivatives, but in practice, they are rare and insignificant, so ignoring them keeps us inside the 10^{-4} relative error.

If $f'(L) = f'(R)$ and $f(R) \neq f(L) + f'(L) \cdot (R - L)$, proceed to the calculation of the integrals at the segments $[L; (L + R)/2]$ and $[(L + R)/2; R]$.

If $f'(L) \neq f'(R)$, then find the x-coordinate where these tangents cross, denote it as M.

If $M \leq L$, or $M \geq R$, or $f(M) \neq f(L) + f'(L) \cdot (M - L)$, proceed to the calculation of the integrals at the segments $[L; (L + R)/2]$ and $[(L + R)/2; R]$.

If $L < M < R$ and $f(M) = f(L) + f'(L) \cdot (M - L)$:

If $f'(L) = f'(M - eps), f'(M + eps) = f'(R)$, then we have found a vertex and can add the area of two trapezoids to the answer. Again, ignore the outliers.

If $f'(L) = f'(M - eps)$ and $f'(M + eps) \neq f'(R)$ or $f'(L) \neq f'(M - eps)$ and $f'(M + eps) = f'(R)$, then add a trapezoid area [again, ignore the outliers] and keep calculating the integral on the other segment.

If $f'(L) = f'(M + eps)$ and $f'(M - eps) = f'(R)$, then we found a vertex, proceed to the calculation at the segments $[L; M - eps]$ and $[M + eps; R]$, and add the areas of the small trapezoids.

In practice, this algorithm causes $2000 - 3000$ iterations.

Problem H. Don't Try This at Home

Compress all the values to the range $[1; m]$, where m is the amount of different values in the array.

In the most generic case, applying the function f is implemented like this:

1. iterate over the indexes from right to left;
2. if the current value is less than m:
3. increase the current value by 1;
4. erase all the values to the right from it;
5. note all the initial values that are missing right now;
6. if their amount is greater than the number of the erased spots, goto 1;
7. else, fill the erased spots in the tail with the missing values in the increasing order, and fill the remaining erased spots with 1. break.

As long as the array contains a non-empty permutation of unique elements at its tail, and it's not sorted in the decreasing order, applying the function f to this array causes this permutation to change to the lexicographically next permutation, and the elements outside of it remain the same. Thus, if there is such a permutation at the array's tail, find the amount of permutations that are lexicographically greater than this one, add this value to the answer, and sort it in the decreasing order.

If that permutation is empty, or sorted in a decreasing order, then applying the function f causes the value directly to the left from that permutation to be increased by 1, and that permutation is sorted in the increasing order again. Thus you have to simulate increasing that value to the left until you get a required array, while adding a factorial each time. In at most $2n$ simulations, you'll find a required array.

You have to think of all the corner cases to be able to perform a simulation in $O(1)$. That includes these corner cases: the value to the left is equal to 1, to m, or to some value in the tail permutation minus one.

However, if that value is equal to m, you may perform that simulation in $O(n)$, as this case appears at most once.

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. Spoiler

For any f_1, k the sequence f is always interpolated as $f_n = \frac{k}{2}n^2 + bn$ or $\frac{k+1}{2}n^2 + bn$ for some fixed integer b starting with some index.

Proof for even k.

Define b_n:

$$b_n = \frac{f_n - \frac{kn^2}{2}}{n}$$

Then:

$$b_{n+1} = \frac{f_{n+1} - \frac{k(n+1)^2}{2}}{n+1} = \frac{k(n+1) + \left\lceil \frac{f_n}{n+1} \right\rceil \cdot (n+1) - \frac{k(n+1)^2}{2}}{n+1} =$$

$$= \frac{k(n+1) + \left\lceil \frac{\frac{kn^2}{2}+b_n n}{n+1} \right\rceil \cdot (n+1) - \frac{k(n+1)^2}{2}}{n+1} = k + \left\lceil \frac{\frac{kn^2}{2} + b_n n}{n+1} \right\rceil - \frac{k(n+1)}{2} =$$

$$= \left\lceil \frac{\frac{kn^2}{2} + b_n n}{n+1} \right\rceil - \frac{k(n-1)}{2} = \left\lceil \frac{\frac{kn^2}{2} - \frac{k(n^2-1)}{2} + b_n n}{n+1} \right\rceil = \left\lceil \frac{\frac{k}{2} + b_n n}{n+1} \right\rceil = \left\lceil \frac{\frac{k}{2} - b_n}{n+1} \right\rceil + b_n.$$

If $b_n > n + \frac{k}{2}$, then $-n - 1 \geq \frac{k}{2} - b_n$, so $\left\lceil \frac{\frac{k}{2}-b_n}{n+1} \right\rceil < 0$, and $b_{n+1} < b_n$.

If $b_n < \frac{k}{2}$, then $\frac{k}{2} - b_n > 0$, so $\left\lceil \frac{\frac{k}{2}-b_n}{n+1} \right\rceil > 0$, and $b_{n+1} > b_n$.

If $n + \frac{k}{2} \geq b_n \geq \frac{k}{2}$, then $-n - 1 < \frac{k}{2} - b_n \leq 0$, so $\left\lceil \frac{\frac{k}{2}-b_n}{n+1} \right\rceil = 0$, and $b_{n+1} = b_n$

Therefore, the sequence of b_n always stabilizes at some constant, and the possible interpolations are $f_n = \frac{k}{2}n^2 + (\frac{k}{2} + m)n$, $f_n = \frac{k}{2}n^2 + (\frac{k}{2} + m - 1)n$, or $f_n = \frac{k}{2}(n^2 + n)$, where m is the interpolation beginning index.

Proof for odd k.

Define b_n:

$$b_n = \frac{f_n - \frac{(k+1)n^2}{2}}{n}$$

Then:

$$b_{n+1} = \frac{f_{n+1} - \frac{(k+1)(n+1)^2}{2}}{n+1} = \frac{k(n+1) + \left\lceil \frac{f_n}{n+1} \right\rceil \cdot (n+1) - \frac{(k+1)(n+1)^2}{2}}{n+1} =$$

$$= \frac{k(n+1) + \left\lceil \frac{\frac{(k+1)n^2}{2}+b_n n}{n+1} \right\rceil \cdot (n+1) - \frac{(k+1)(n+1)^2}{2}}{n+1} = k + \left\lceil \frac{\frac{(k+1)n^2}{2} + b_n n}{n+1} \right\rceil - \frac{(k+1)(n+1)}{2} =$$

$$= \left\lceil \frac{\frac{(k+1)n^2}{2} + b_n n}{n+1} \right\rceil - \frac{(k+1)(n-1)}{2} - 1 = \left\lceil \frac{\frac{(k+1)n^2}{2} - \frac{(k+1)(n^2-1)}{2} + b_n n}{n+1} \right\rceil - 1 = \left\lceil \frac{\frac{k+1}{2} + b_n n}{n+1} \right\rceil - 1 =$$

$$= \left\lceil \frac{\frac{k+1}{2} - b_n}{n+1} \right\rceil + b_n - 1.$$

If $b_n > \frac{k-1}{2}$, then $0 \geq \frac{k+1}{2} - b_n$, so $\left\lceil \frac{\frac{k+1}{2}-b_n}{n+1} \right\rceil < 1$, and $b_{n+1} < b_n$.

If $b_n < \frac{k-1}{2} - n$, then $\frac{k+1}{2} - b_n > n + 1$, so $\left\lceil \frac{\frac{k+1}{2}-b_n}{n+1} \right\rceil > 1$, and $b_{n+1} > b_n$.

If $\frac{k-1}{2} \geq b_n \geq \frac{k-1}{2} - n$, then $0 < \frac{k+1}{2} - b_n \leq n + 1$, so $\left\lceil \frac{\frac{k+1}{2}-b_n}{n+1} \right\rceil = 1$, and $b_{n+1} = b_n$

Similarly to the case with an even k, the sequence of b_n always stabilizes at some constant, and the possible interpolations are $f_n = \frac{k+1}{2}n^2 + (\frac{k-1}{2} - m)n$, $f_n = \frac{k+1}{2}n^2 + (\frac{k}{2} - m + 1)n$, or $f_n = \frac{k+1}{2}n^2 + \frac{k-1}{2}n$, where m is the interpolation beginning index.

It can be also proven the interpolation beginning index does not exceed $2 \cdot 10^5$ under the conditions on f_1, k given in the statements. Thus the solution is to iterate over all the divisors of x smaller than $2 \cdot 10^5$, try to get k from each of six equations, and if it is integer, try to find the fitting f_1 with the binary search: the bigger is $|f_1 - k|$, the bigger the interpolation beginning index is.