PROBLEM:

There are N bags, each containing M balls numbered 1 to M.
You will choose exactly one ball from each bag, and arrange them randomly in a row.

Find the number of arrangements such that:

- There exists a ball labelled K appearing in this row, such that the number of balls appearing before it with values $\geq K$, is $< K$.

EXPLANATION:

There are a couple of different solutions to this task.

Solution 1

Let's look at how the balls can be chosen, in terms of which have 'large' labels and which have 'small' ones.

Out of the N balls, suppose x of them have values $\geq K$.
The positions of these x balls can be chosen in $\binom{N}{x}$ ways.
Then,

- The other $N - x$ positions can take *any* values less than K.
  - This gives a total of $(K-1)^{N-x}$ choices.
- For the x positions with values $\geq K$, we need to ensure that:
  - At least one of them contains the value K.
  - The leftmost occurrence of K is somewhere within the first K positions; since that will ensure that this leftmost K satisfies the condition of having $< K$ elements that are $\geq K$ before it.

The second condition is a bit annoying to deal with directly, so in this case, we can instead count the number of *bad* arrangements and subtract them from the total number of arrangements.
That is, we'll count the number of arrangements that either don't contain K at all, or which contain K but not within the first K indices.
You may verify that:

- The total number of choices is $(M-K+1)^x$, freely assigning something between K and M to each index.
- The number of arrangements in which K isn't present at all is $(M-K)^x$.
- For K to be present but not in the first K indices,
  - The first K indices should have values $> K$, for $(M-K)^K$ choices in total.
  - Then, we again count the total number of choices for all other indices and subtract from that the count of ways where no K is present: namely, $(M-K+1)^{x-K} - (M-K)^{x-K}$.
  - Note that this case doesn't need to be considered when $x \leq K$.

All together, the answer we get is

$$\sum_{x=0}^{K} \binom{N}{x}(K-1)^{N-x} \cdot ((M-K+1)^x - (M-K)^x)$$
$$+ \sum_{x=K+1}^{N} \binom{N}{x}(K-1)^{N-x} \cdot ((M-K+1)^x - (M-K)^x - (M-K)^K \cdot ((M-K+1)^{x-K} - (M-K)^{x-K}))$$

For a fixed x this is just a binomial coefficient and a few powers, so we get a solution in $\mathrm{O}(N \log N)$ time (or even linear, if you precompute powers).
Solution 2

Let's fix the leftmost occurrence of K, say at index i. Then,

- Everything after i can be chosen freely, for M^{N-i} choices.
- K can't occur before i, and we need to ensure that there are less than K elements before i that are $> K$.

To deal with the latter, let's fix x, the number of elements $> K$ before i.
Then,

- Their positions can be chosen in $\binom{i-1}{x}$ ways.
- Each of these x indices has $M - K$ choices for its value, for $(M-K)^x$ ways.
- The other indices before i have values $< K$, for $(K-1)^{i-1-x}$ ways in total.

So, our answer is

$$\sum_{i=1}^{N} (M^{N-i} \sum_{x=0}^{K-1} \binom{i-1}{x} (M-K)^x (K-1)^{i-1-x})$$

This is quadratic, of course, but can be sped up surprisingly easily: use Pascal's identity!
Specifically, define $S_i = \sum_{x=0}^{K-1} \binom{i-1}{x} (M-K)^x (K-1)^{i-1-x}$, so that we're trying to compute $\sum_{i=1}^{N} M^{N-i} S_i$.

Now, apply Pascal's identity, so that $\binom{i-1}{x} = \binom{i-2}{x} + \binom{i-2}{x-1}$.
Substituting this into the expression for S_i, you can see that S_i can be computed in terms of S_{i-1} instead.

Specifically, we have

$$S_i = S_{i-1} \cdot (K-1) + (S_{i-1} - \binom{i-2}{k-1} (M-K)^{K-1} (K-1)^{i-1-K}) \cdot (M-K)$$

This way, all the S_i values can be computed in $\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$, and we're done.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

For an array A, let $f(A) = 1$ if it can be fully deleted using the following operation (and $f(A) = 0$ otherwise):

- Choose two adjacent unequal elements of A and delete them.
  The remaining elements are concatenated together.

Let $g(A)$ denote the sum of $f(A)$ across all non-empty subarrays of A.
Given N and M, compute $\sum g(A)$ across all arrays of length N containing integers from 1 to M.

EXPLANATION:

Our first order of business should be to work out what $f(A)$ is for a single array A; after all, everything else in the task hinges on it.

So, when is it possible to delete all the elements of an array with the given process?
Obviously, if the array has odd length it's impossible; so we only look at even-length arrays.
Intuitively, if the same element appears too many times then this won't be possible; and otherwise it's likely possible.
In particular, notice that each move can only delete one copy of an element - and we need to make $\frac{N}{2}$ moves in total for an array of length N. This tells us the following:
Claim: Let A be an array of even length N. $f(A) = 1$ if and only *every* element of A appears at most $\frac{N}{2}$ times.

Proof

If an element appears $> \frac{N}{2}$ times, as noted above at least one of its occurrences will remain after $\frac{N}{2}$ moves - meaning the array cannot be empty.

Now, suppose every element appears $\leq \frac{N}{2}$ times.
Let x be the element with maximum frequency (i.e, a mode of A); and F be this maximum frequency.
Then,

- If $F = \frac{N}{2}$ and there are two different elements with frequency F, delete one copy of each of them (there will definitely be such a pair that's adjacent to each other).
- If $F = \frac{N}{2}$ and the mode is unique, delete one copy of x and any other element.
- If $F < \frac{N}{2}$, delete any two adjacent different elements.

Note that these cases are exhaustive since $F \leq \frac{N}{2}$; and after each move, we've reduced the length of the array by 2 while also ensuring that in the new array, no element has frequency $> \frac{N-2}{2}$.
Repeatedly performing this process will empty the array, as desired.

Now that we know about $f(A)$, let's turn to $g(A)$.
$g(A)$ is computed as the sum of $f(A)$ across all subarrays of A; but as noted above it's enough to only consider even-length subarrays.
We want the sum of $g(A)$ across all possible arrays A of length N with elements from 1 to M.

Observe that the subarrays are pretty independent here, allowing us to use the technique of contribution.
That is, instead of fixing A and trying to compute $g(A)$; we'll fix a subarray S of A such that $f(S) = 1$, and then compute the number of arrays A for which S is a subarray - summing this up across all S will give us the same answer.

So, let's fix the endpoints L and R of the subarray, and try to count the number of arrays A in which $A[L \ldots R]$ adds 1 to the total.

Notice that:

- The exact values of L and R don't matter much: only the length $(R - L + 1)$.
- Everything outside the subarray can be freely chosen; and each element has M independent options.

In particular, suppose we knew for each length k the value $c_M(k)$: the number of arrays of length k with elements from 1 to M, such that they can be fully deleted.
Then, the overall answer is just

$$\sum_{k=1}^{\frac{N}{2}} c_M(2k) \cdot (N - 2k + 1) \cdot M^{(N-2k)}$$

This is because we can choose any deletable array of length $2k$, choose its starting position, and choose the values of all the other elements.

All that remains is to compute the $c_M(k)$ values, and we'll be done!

Computing $c_M(k)$

Recall that we have a condition for a deletable array of length $2k$: every element must have frequency $\leq k$.
This is a bit unwieldy to deal with, so let's do the opposite: we'll count arrays of length $2k$ that *can't* be deleted entirely, then subtract them from the total number of arrays (which is just M^{2k}).
Let's redefine $c_M(k)$ to be this value, i.e, arrays that can't be deleted.

For an array that can't be deleted entirely, some element must have frequency $> k$.
Of course, exactly one element can have frequency $> k$.
So,

- Fix the element that has frequency $> k$ (M choices here).
- Fix this frequency that's $> k$ (anything between $k + 1$ and $2k$, say x).
- Fix the x positions that this element appears in ($\binom{2k}{x}$ choices).
- Finally, fix the values at the other positions.
  With $M - 1$ choices per position, that's $(M - 1)^{2k-x}$ choices in total.

All together, that tells us that

$$c_M(2k) = \sum_{x=k+1}^{2k} M \cdot \binom{2k}{x} \cdot (M - 1)^{2k-x}$$

This is correct, but of course computing this directly for all $2k$ will take $\mathrm{O}(N^2)$ time.

Optimizing the computation

Let's look at the expression we have.

$$c_M(2k) = \sum_{x=k+1}^{2k} M \cdot \binom{2k}{x} \cdot (M-1)^{2k-x}$$

M is a constant independent of x here, so for convenience, let's take it out and multiply it back in at the end.
We'll also reindex the summation a bit, to get

$$\sum_{x=0}^{k-1} \binom{2k}{x}(M-1)^x$$

This uses the fact that $\binom{N}{K} = \binom{N}{N-K}$ (alternately, you can see it as the expression obtained by fixing the positions and values of the small-frequency elements).

Let's relax the conditions a bit: instead of considering only even-length arrays, we'll allow for any length - after all, we're just counting the number of arrays such that some element appears more than half the time.
So, let

$$f_M(k) = \sum_{x=0}^{2x<k} \binom{k}{x}(M-1)^x$$

As it turns out, we can compute $f_M(k+1)$ from $f_M(k)$.
There are minor differences when doing this for even and odd k, so I'll present odd k below and even k will be left as an exercise for the reader.
We're trying to compute $f_M(2k+2)$ from $f_M(2k+1)$. Observe that:

$$\begin{aligned}
f_M(2k+2) &= \sum_{x=0}^{k} \binom{2k+2}{x}(M-1)^x \\
&= \binom{2k+2}{0}(M-1)^0 + \binom{2k+2}{1}(M-1)^1 + \ldots + \binom{2k+2}{k}(M-1)^k \\
&= \left(\binom{2k+1}{0} + \binom{2k+1}{-1}\right)(M-1)^0 + \left(\binom{2k+1}{1} + \binom{2k+1}{0}\right)(M-1)^1 + \ldots + \left(\binom{2k+1}{k} + \binom{2k+1}{k-1}\right)(\\
&= f_M(2k+1) + (M-1) \cdot f_M(2k+1) - \binom{2k}{k}(M-1)^{k+1}
\end{aligned}$$

The third line arises from simply applying [Pascal's rule]() to each binomial coefficient.
Note that $\binom{x}{-1} = 0$ for any $x \geq 0$.

In this way, $f_M(2k+2)$ can be computed from $f_M(2k+1)$ in constant time.
A similar computation allows you to get $f_M(2k+1)$ from $f_M(2k)$ in constant time.
So, starting with $f_M(2) = 1$, you can get every $f_M(k)$ value in $\mathrm{O}(N)$ time in total.

Once these are known, the $c_M(2k)$ values are trivially found (since $c_M(2k) = f_M(2k) \cdot M$), after which the final answer is easily computed in linear time as well.

If you aren't comfortable with algebraic manipulation like we did towards the end, there's a more visual way to get to the result (though the formula ends up the same either way).

I refer you to the editorial of AtCoder ABC 235G 9 for more details on this.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

An $N \times N$ matrix is called K-strange if all its elements are distinct numbers from 1 to N, and $(A_{i,j} - A_{j,i})$ is a multiple of K for all (i,j).
Let M_N be the largest number such that there exists a M_N-strange $N \times N$ matrix.

You're given a matrix A with some elements missing.
Find the number of ways to fill in the missing elements such that the resulting matrix is M_N-strange.

EXPLANATION:

There's a lot of stuff going on in the statement, so a good first step is to take things one at a time.
Our solution process can be broken up into three steps:

- First, find M_N for a given N.
- Second, characterize all M_N-strange matrices.
- Finally, find out how many of them can be obtained from A.

Finding M

We want to find the largest integer for which there exists a strange matrix.
Suppose we fix K. When can there be a K-strange matrix?

Notice that:

- $A_{1,2}$ and $A_{2,1}$ should be the same, modulo K. This means $A_{1,2} - A_{2,1}$ must be a multiple of K.
  Since they must be distinct integers, their values must differ by at least K - that is, $\min(A_{1,2}, A_{2,1}) + K \leq \max(A_{1,2}, A_{2,1})$.
- In fact, the exact same reasoning applies to any pair of elements (i,j) with $i < j$: the pair $(A_{i,j}, A_{j,i})$ should have a difference of at least K between them.

There are $\frac{N \cdot (N-1)}{2}$ such pairs of opposite elements.
Since everything must be distinct, this means there will definitely be some (i,j) such that $\min(A_{i,j}, A_{j,i}) \geq \frac{N \cdot (N-1)}{2}$.

Since the larger element of this pair should be at least K away from the smaller one, the larger element is *at least* $K + \frac{N \cdot (N-1)}{2}$.
However, this larger element should also be within N^2, giving us the inequality $K + \frac{N \cdot (N-1)}{2} \leq N^2$.

This immediately gives us an upper bound on K: $K \leq N^2 - \frac{N \cdot (N-1)}{2}$.

With a bit of trial and error, it's not hard to see that you can also construct a matrix that satisfies this K.
For example, with $N = 4$, we get $K = 4 \cdot 5/2 = 10$, and one valid matrix is:

$$\begin{bmatrix} 7 & 11 & 12 & 14 \\ 1 & 8 & 13 & 15 \\ 2 & 3 & 9 & 16 \\ 4 & 5 & 6 & 10 \end{bmatrix}$$

More generally: you can fill the numbers from 1 to $N \cdot (N-1)/2$ in the lower triangle of the matrix; then add $K = N \cdot (N+1)/2$ to each of them and place that value in the symmetric part; and finally place all remaining elements in

the middle.

We thus have an upper bound for K, and also a way to achieve this upper bound.
So, $M_N = N \cdot (N+1)/2$
Which matrices?

Next, let's figure out which matrices can possibly be M_N-strange.
We know $M_N = N \cdot (N+1)/2$.
It can be observed that:

- If some integer x is such that $x + M_N > N^2$ and $x - M_N < 1$; this integer can *only* be on the diagonal of the matrix (since it can't be paired to anything else).
  There are exactly N such integers: $M_N, M_N - 1, M_N - 2, \ldots, M_N - N + 1$.
  Coincidentally, there are N diagonal spots too.
  So, the diagonal should consist of exactly these elements, in some order!
- Now, let's look at the values other than these N.
  Since M_N is so large, we can see that all these elements will "pair up"; forming the pairs $(1, M_N + 1), (2, M_N + 2), (3, M_N + 3), \ldots, (N \cdot (N-1)/2, N^2)$.
- For each of these pairs, one element should be in the lower triangle, and the other in the upper triangle in the symmetric position.
  However, it doesn't matter where which pair is placed, since they're independent of each other.

Filling in A

Now, we have a good idea of what a M_N-strange matrix should look like. Let's see how many ways A can be filled in to satisfy this.

First, let's look at the elements that must be on the diagonal - $M_N, M_N - 1, M_N - 2, \ldots, M_N - N + 1$.

- If any of them exist outside the diagonal, the answer is 0.
- If any other elements are on the diagonal, the answer is 0.
- Otherwise, suppose k of them are on the diagonal, and $N - k$ don't appear anywhere else in the matrix.
  These $N - k$ elements can be arranged in any order along the diagonal, for $(N-k)!$ ways in total.

Next, let's look at the other elements.
Consider some pair $(x, x+K)$ where $x \le N \cdot (N-1)/2$.

- If x and $x + K$ both appear in the matrix, they must be opposite each other; otherwise the answer is 0.
- If exactly one of x and $x + K$ appears, the value of the cell opposite to it is uniquely fixed (to the other element of the pair).

Now, suppose there are c pairs such that neither x nor $x + K$ appear in A.
Then,

- There will also be exactly c pairs of opposite cells that are empty; everything else will be filled (since if one cell of a pair was originally filled, the other is uniquely determined).
- We can assign there c pairs of values to the c empty cells in any order; for $c!$ ways.
- Further, for each pair we can choose whether the smaller or the larger value appears in the bottom triangle.
  With 2 choices for each pair, we get 2^c choices.

So, the final answer is

$$(N-k)! \times c! \times 2^c$$

where:

- k diagonal elements are already present; and
- c pairs of values don't appear in A at all.

Of course, this is only after checking if the answer is 0.

Note that you need to calculate some factorials and powers of two, but that can be done in a bruteforce fashion since $N - k$ and c are both $\leq N^2$; and the sum of N^2 is bounded.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ per testcase.

PROBLEM:

You have N binary strings, each of length M. The i-th is S_i.
You can do the following:

- Choose a triple (i, j, k) such that $1 \le i < N$, $1 \le j < k \le M$, and $S_{i+1,j} = S_{i+1,k} = 1$.
- Then, swap $S_{i,j}$ and $S_{i,k}$.

Find the number of distinct strings S_1 attainable via this process.

EXPLANATION:

Let's start from the bottom of the grid and work our way upwards.

S_N can't be changed at all, so there's only 1 possibility for it.
Next, let's look at S_{N-1}.
Note that:

- If $S_{N,i} = 0$, then $S_{N-1,i}$ can't be moved, and is fixed.
- On the other hand, if $S_{N,i} = 1$, then $S_{N-1,i}$ can be swapped to some other position where S_N contains a 1.
  In fact, if we look at only positions where S_N contains a 1, the values at these positions in S_{N-1} can be freely rearranged since any pair among them can be swapped.

Next, look at S_{N-2}.
Once again, the elements at positions that are 1's in S_{N-1} can be freely moved around.
However, there's more: the power to rearrange some part of S_{N-1} gives us the ability to choose which set of positions contain 1's in S_{N-1} (to some extent).

In particular, let's define P_i to be the set of indices that can be freely rearranged in S_i.
If an index is not in P_i, that means it's fixed.
For instance, P_N is empty (because all the elements of S_N are fixed), and P_{N-1} consists of all those indices i such that $S_{N,i} = 1$.

Let's try to compute P_{N-2} given that we know P_{N-1}.

- First, if S_{N-1} contains zero or one occurrence of 1, P_{N-2} will be empty: after all, there's no way to rearrange the string.
- Otherwise, P_{N-2} will definitely consist of, at the very least, all those indices that are 1 in S_{N-1}.
  However, as noted earlier, it might be larger: some one those ones in S_{N-1} can be moved to different indices depending on the contents of P_{N-1}.
- In particular, we have the following:
  - If there exists an index $i \in P_{N-1}$ such that $S_{N-1,i} = 1$, then P_{N-2} will contain *all* of P_{N-1} (in addition to any indices that contain ones).
    This is because this 1 can be shuffled around among the indices of P_{N-1}, which gives us enough freedom with swaps.
  - If no such $i \in P_{N-1}$ exists, P_{N-2} will consist of only those indices in P_{N-1} that contain ones.

So, knowing P_{N-1} and S_{N-1}, it's quite easy to compute P_{N-2} in $\mathrm{O}(M)$ time: the only thing we need to check is whether there's any "intersection" between P_{N-1} and ones in S_{N-1}.

Simply repeat this process over and over again, till you finally compute P_1: the set of free positions in S_1.
Now, we know that:

- Indices not in P_1 have their values fixed.
- Indices in P_1 can have their values freely shuffled around.
  If there are x zeros and y ones, we can thus obtain $\binom{x+y}{x}$ distinct strings, by choosing which positions the zeros occur at.

Computing the final binomial coefficient will require division, which in the presence of a modulo can be done with the help of multiplicative inverses.

TIME COMPLEXITY:

$\mathrm{O}(N \cdot M)$ per testcase.

PROBLEM:

For a permutation P of $\{1, 2, \ldots, N\}$, you can perform the following operation exactly once:

- Partition P into some number of subarrays, say K subarrays.
  Let M_i denote the maximum element of the i-th subarray.
- Delete the $\lfloor \frac{K}{2} \rfloor$ subarrays with smallest M_i values.
  In particular, if $K = 1$ nothing gets deleted.

Find the number of distinct resulting arrays.

EXPLANATION:

Let's count the number of subsets of elements that can be deleted via the above process - this is equivalent to counting the number of final arrays.

Let M denote the *largest* deleted element, meaning we can only delete elements $\leq M$.
Further, since our process only allows us to delete subarrays, we can only delete some disjoint subarrays of P that all contain only elements $\leq M$.

Let the positions of elements $> M$ in P be $i_1 < i_2 < i_3 < \ldots < i_{N-M}$.
Note that:

- Any deleted subarray must lie fully between $i_j + 1$ and $i_{j+1} - 1$ for some j, or be a prefix/suffix of P.
- Further, from a fixed range $[i_j + 1, i_{j+1} - 1]$, at most one subarray can be chosen for deletion.

So, we can pick upto $N - M + 1$ subarrays to be deleted - one from each maximal segment between i_j's, and one prefix and suffix.
Observe that if we pick $N - M + 1$ subarrays to be deleted, it isn't actually possible to delete them all: each element $> M$ can be the maximum of at most one non-deleted subarray, meaning we have at most $N - M$ non-deleted subarrays; so deleting $N - M + 1$ isn't possible.

On the other hand, if we pick $K \leq N - M$ subarrays, it's always possible to find a partition that deletes exactly them: it's not hard to see how (start with $N - M$ separate subarrays, each containing one of the elements $> M$, after which some of them can be merged if necessary).

So, all we need to do is find the number of ways of picking $\leq N - M$ non-empty subarrays, with at most one from each maximal segment between i_j's, and with M included in one of them.

To do that, we'll compute all ways of choosing $N - M + 1$ subarrays (some of which are possibly empty), and from them subtract the ways of choosing $N - M + 1$ non-empty subarrays.

Counting all ways

Let $L_j = i_j - i_{j-1} - 1$ be the length of the j-th maximal segment, along with $L_1 = i_1 - 1$ and $L_{N-M+1} = N - i_{N-M}$.
Then,

- As noted earlier, from L_1 we can only choose some prefix.
  This gives us $L_1 + 1$ options (recall that we're allowing empty choices here).
- Similarly, we have $L_{N-M+1} + 1$ choices for the suffix.
- Then, for every other L_j, we have $\frac{L_j \cdot (L_j + 1)}{2} + 1$ ways: all possible non-empty subarrays, and the empty subarray.

There is one exception: one of these segments will contain M, and we need to ensure that the subarray chosen in this segment includes M.
That can be done with simple combinatorics, though do remember to account for the prefix/suffix cases appropriately.

The total number of choices is then just the product of all these numbers.
Counting non-empty ways

Exactly the same as above, just that you don't include the $+1$ for choice of empty subarray anymore.
That is,

- L_1 choices for the prefix.
- L_{N-M+1} choices for the suffix.
- $\frac{L_j \cdot (L_j+1)}{2}$ choices for everything else.

The computation for subarrays including M doesn't change, since that's non-empty anyway.

Once these numbers are known, say X_M and Y_M respectively, simply add $(X_M - Y_M)$ to the answer.

The above discussion gives us a solution in $\mathrm{O}(N^2)$: fix a value of M, then find X_M and Y_M by computing the lengths of empty segments and multiplying appropriate numbers.

To speed it up, consider what happens when M is processed in descending order.
When moving from M to $M-1$, the segments stay largely the same: the only change is the segment with $M-1$, which will break into two smaller segments.

Since the values we're looking for are products depending only on the lengths of the segments, they don't change much either: you can simply divide out the old value of this segment, then multiply in the values of the new segments.
There are various ways of actually maintaining the segments. For instance:

- Keep the segments themselves in a set, and find which one to break with binary search; or
- Note that when a segment (say with maximum M) breaks, the new segments depend only on the position of M, and the next/previous greater elements of M in P, which can be precomputed.

Alternately, you can process M in increasing order, merging segments as you go - for instance with a DSU (or, again, a set).

TIME COMPLEXITY:

$\mathrm{O}(N \log MOD)$ per testcase.

PROBLEM:

Given an array A of length N and a fixed parameter K, you can do the following:

- Choose any integer X, and delete every element of A that lies between X and $X+K-1$.

Let $f(A)$ denote the minimum number of such moves needed to delete every element of A.
Given N, M, K, compute the value of $f(A)$ across all strictly increasing arrays of length N with elements from 1 to K.

EXPLANATION:

First, let's compute $f(A)$ for a single array.
This is quite simple: it can be seen that the optimal strategy is to just keep repeatedly choosing X to be the minimum element of A till everything's gone.

In particular, we'll always need $\leq \lceil \frac{M}{K} \rceil$ moves to delete any array A with elements between 1 and M.

Let's fix $f(A) = c$, and try to find the number of arrays that require exactly c moves to delete.
As observed earlier, this means we pick the minimum remaining element of A exactly c times.

Suppose these minimums are $m_1 < m_2 < \ldots < m_c$.
Then, the following conditions must hold:

- $1 \leq m_i \leq M$ for each $1 \leq i \leq c$.
- $m_i + K \leq m_{i+1}$ for each $1 \leq i < c$.
  This is to ensure that when picking m_i, m_{i+1} doesn't get deleted.

It's not too hard to count the number of ways of choosing these m_i.
If we let $d_i = m_i - m_{i-1}$ (with $m_0 = 0$ and $m_{c+1} = M$), choosing the m_i is equivalent to assigning values to the d_i such that:

- The sum of all d_i is exactly M (we start at 0 and continually add differences till we reach M)
- $d_1 > 0$
- $d_i \geq K$ for each $2 \leq i \leq c$
- $d_{c+1} \geq 0$

Counting the number of solutions to such a system is a classical stars-and-bars task: letting $d_1' = d_1 - 1$ and $d_i' = d_i - K$ for $2 \leq i \leq c$, we just want the number of solutions to $c+1$ non-negative integers summing up to $M - 1 - (c-1)K$.

What about the other elements, i.e, the ones that aren't chosen as the minimums?
There are $N - c$ of them, but notice that they must be some specific elements.
In particular, these $N - c$ elements should be of the form $m_i + x$, where $1 \leq x < K$ - that's how we ensure that they are indeed deleted when the m_i are picked.

Suppose there are a total of L elements of this form.
We can choose any $N - c$ of them, for a total of $\binom{L}{N-c}$ ways. All we need to do, is actually find L.
That isn't too hard:

- For each of $m_1, m_2, \ldots, m_{c-1}$ there are $(K-1)$ such choices.

- For m_c, there are $K-1$ choices if $m_c + K - 1 \le M$, and $M - m_c$ choices otherwise.
  - So, if $m_c \le M + 1 - K$, we have $L = c \cdot (K-1)$
  - Otherwise, we have $L = (c-1) \cdot (K-1) + M - m_c$

Note that the stars-and-bars part also needs to be modified slightly to account for these cases.

- If $m_c \le M + 1 - K$, we're essentially asking for $d_{c+1} \ge K - 1$, rather than $d_{c+1} \ge 0$
- Otherwise, we force $d_{c+1} = M - m_c$.
  However, observe that this happens only for K values of m_c, and so can be bruteforced: since the number of moves is $\le \lceil \frac{M}{K} \rceil$, the total complexity remains $\mathrm{O}(M)$.

TIME COMPLEXITY:

$\mathrm{O}(M)$ per testcase.

PROBLEM:

For a permutation P of length M, let $F(P, K)$ denote the number of sorted arrays of length K with elements from 1 to K, such that

$$\min_{j=1}^{A_1} P_j = \min_{j=1}^{A_i} P_j$$

for every $2 \le i \le K$.

Given N, K, V find the sum of $F(P, K)$ across all permutations P of $\{1, 2, \ldots, N\}$ such that $P_1 = V$.

EXPLANATION:

The condition $\min_{j=1}^{A_1} P_j = \min_{j=1}^{A_i} P_j$ for a sorted array A means that the prefix minimum of P till index A_1 should equal the prefix minimum of P till index A_K; the rest will be satisfied automatically.
In particular, this means that if M is the prefix minimum till A_1, the elements $1, 2, 3, \ldots, M-1$ can appear only *after* A_K.
Since $P_1 = V$ is fixed, the prefix minimum till A_1 also cannot exceed V, no matter what A_1 is.

Let's fix M, the prefix minimum till A_1, and then try to count all sorted arrays A (and permutations they belong to) that are satisfied by M.

This is rather hard if approached directly — the trick is to look at *differences*.
Observe that any sorted array can be represented in terms of the differences of its adjacent elements (as long as at least one of the elements is known, of course).

So, let's try to represent *everything* in terms of differences; or rather distances.
We know $P_1 = V$, which can't be changed.
Let d_1 denote the distance between index 1 and the position of M, the minimum.
d_1 can be any **positive** integer.

After M's position is fixed, we have the array A. Note that all the elements of A must lie to the right of the position of M.
So, we can have distances $d_2, d_3, \ldots, d_K$, where:

- d_2 is the distance between the position of M and A_1.
- For $3 \le i \le K+1$, d_i is the difference between A_{i-2} and A_{i-1}.

With this, everything upto A_K is determined by choosing values for the d_i.
Note that each of d_2, d_3, d_{K+1} is a **non-negative** integer.
Next, recall that the elements $1, 2, 3, \ldots, M-1$ must be placed after A_K.

So, let's first fix their order (in $(M-1)!$ ways, of course), then try to once again use distances to define their positions.
if you've been following so far, this is straightforward: we need to choose $M-1$ distinct indices after A_K, which can just be modelled as having $M-1$ positive distances.
We can denote these by $d_{K+2}, d_{K+3}, \ldots, d_{K+M}$; all of which are **positive**.

This pretty much completes our model - notice that once we choose values for the d_i, the positions of everything *and* the array A are determined.
We just need to take care of a couple more things:

- We impose the constraint $d_1 + d_2 + d_3 + \ldots + d_{K+M} \leq N - 1$, because the final index we choose should be within the array (we start from index 1 and add differences; which is why the sum is bounded by $N - 1$ and not N).
- The order of elements other than $1, 2, \ldots, M$ and V hasn't been decided; but also really doesn't matter since it doesn't affect A or the prefix minimums of A.
  So, we can choose any of the $(N - M - 1)!$ orders for them.

We're almost done.
All we need to do is count the number of solutions to $d_1 + d_2 + d_3 + \ldots + d_{K+M} \leq N - 1$ and multiply by $(M - 1)! \cdot (N - M - 1)!$ to get the total contribution of M.

If we had *equality* instead of an *inequality*, this wouldn't be so hard: the number of solutions can be found using stars-and-bars.
As-is, we could enumerate all $\mathrm{O}(N)$ choices for what the sum should be and find the answer, but there's a quicker way.
Let's add a *slack variable*, d_{M+K+1}, which denotes how far away from $N - 1$ the sum is.

Then, all we need to do is count the number of solutions to

$$d_1 + d_2 + \ldots + d_{M+K+1} = N - 1$$

which is a single application of stars-and-bars!

Be careful though: some of the d_i must be positive while others can be non-negative.
Stars-and-bars applies only when they're all of the same type, so add/subtract 1 from both sides appropriately so get an equation where all variables are of the same type.
The required binomial coefficient comes out to be $\binom{N+K-1}{M+K}$.

The above solved a single $M < V$.
For $M = V$, almost the same analysis holds: the only difference is that we no longer need the variable d_1 (since V is always fixed to the first position), and the number of "unfixed" elements is $(N - M)$ instead of $(N - M - 1)$ (since the previous case had V *and* $1, 2, \ldots, M$ fixed but we have $V = M$).
The process is still the same though, and you'll get a similar-looking binomial coefficient.

Now that a single M can be solved in $\mathrm{O}(1)$ time, just iterate over them all and add up the answers to get a solution in $\mathrm{O}(N)$ (or rather, $\mathrm{O}(V)$ if necessary stuff is precomputed).

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

You're given a tree on N vertices, rooted at 1. Each vertex should be colored either red or blue.
How many colorings exist such that in *every* subtree, the difference between the number of red and blue vertices is at most 1?

EXPLANATION:

We have a counting problem on a tree, with conditions that every subtree needs to satisfy — which screams at us to try dynamic programming.

Let $dp(u, k)$ denote the number of ways of coloring the subtree of u such that the number of red vertices minus the number of blue vertices is *exactly* k.
Note that k is allowed to be negative here.

Since we only care about the differences being ≤ 1, it's enough to consider $-1 \leq k \leq 1$.
Further, observe that $dp(u, 1) = dp(u, -1)$ for every u. This arises from the fact that given a coloring with one extra red vertex, we can flip the colors of all vertices to obtain a coloring with one extra blue vertex; and vice versa.
So, it's enough to consider the states $k = 0$ and $k = 1$.
We now have $2N$ states, which is small enough. Let's attempt to actually compute their values.

Let $\text{sub}[u]$ be either 0 or 1, denoting the parity of the number of vertices in the subtree of u.
This is easily computed with a DFS.
Observe that if $\text{sub}[u] = 0$, the only possibility is to have a difference of 0, i.e, $dp(u, 1) = 0$ will hold. This is because the difference between the number of red and blue vertices will itself be even, and being ≤ 1 means it can only be 0.
Conversely, if $\text{sub}[u] = 1$, $dp(u, 0) = 0$ will be true.
(This also means it's enough to use N states rather than $2N$).

Consider a vertex u. Let its children be $v_1, v_2, \ldots v_x$ and $w_1, w_2, \ldots, w_y$; where $\text{sub}[v_i] = 0$ for every i and $\text{sub}[w_i] = 1$ for every i.
Observe that all the subtrees corresponding to v_i *must* contain an equal number of red and blue vertices; so they don't affect the difference at all.
So, any valid arrangement within those subtrees can be chosen, with the total number of ways being the product of all $dp(v_i, 0)$.

Next, let's look at the w_i, the vertices with odd subtree size. There are two cases.

- Suppose $y = 2m + 1$, i.e, there's an odd number of odd children.
  Then, $\text{sub}[u] = 0$ (since u itself counts too), so we compute $dp(u, 0)$.
  How do we get a difference of 0?

Answer

The color of u itself contributes either 1 or -1 to the difference, so the odd children together should contribute -1 or $+1$, respectively.

That is, of the $2m + 1$ odd children, either m of them contribute $+1$ and the rest -1, or vice versa (note that making this choice uniquely fixes the color of u as well).
As noted earlier, $dp(w_i, 1) = dp(w_i, -1)$ for everything, so after choosing which vertices are $+1/-1$, the contribution is the product of all $dp(w_i, 1)$.

This is independent of the actual choice of m vertices, so the overall contribution is
$2 \cdot \binom{2m+1}{m} \cdot \prod dp(w_i, 1)$

- Suppose $y = 2m$, i.e, there's an even number of odd children.
  Then, $\text{sub}[u] = 1$, so we compute $dp(u, 1)$.
  How do we get a difference of 1?

Answer

This can be done just as the previous case was.
The odd children must give a total difference of either 0 or $+2$, which then uniquely determines the color of u.

A difference of 0 can be obtained by choosing an equal number of $+1$'s and -1's, so there are $\binom{2m}{m}$ ways.
A difference of 2 requires choosing 2 more $+1$'s then -1's, which can be done in $\binom{2m}{m+1}$ ways.
So, the answer is
$(\binom{2m}{m} + \binom{2m}{m+1}) \cdot \prod dp(w_i, 1)$

Binomial coefficients can be computed quickly by precomputing factorials, after which our solution runs in $\text{O}(N)$.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PROBLEM:

You're given an array A.
Count the number of quadruples (L_1, R_1, L_2, R_2) such that:

- $1 \le L_1 \le R_1 < L_2 \le R_2 \le N$
- $\gcd(A_{L_1}, A_{L_1+1}, \ldots, A_{R_1}) \times \gcd(A_{L_2}, A_{L_2+1}, \ldots, A_{R_2})$ is a perfect square.

That is, count the number of pairs of disjoint subarrays, the product of whose GCDs is a square.

EXPLANATION:

It is a well-known fact that if one end of a subarray (say i) is fixed, there are $\mathrm{O}(\log A_i)$ distinct subarray GCDs among subarrays that start/end at i, since each successive distinct GCD must be a divisor of the previous one (and hence at least halves).
This also tells us that each GCD occurs on some range of other endpoint.

Further, they can all also be computed in $\mathrm{O}(\log N \log A_i)$ time, either using the method here 9 (point 3), or with binary search and some range-gcd structure.

Let st_i denote the distinct GCDs of subarrays (along with the lengths of the corresponding segments) that start at i, and en_i denote the same but for subarrays ending at i.

Suppose we fix R_1, the right endpoint of the first subarray.
Let's also fix the gcd g_1 of that subarray, which as mentioned above gives us some range for L_1.
We want to find the number of subarrays (L_2, R_2) with GCD g_2, such that $L_2 > R_1$ and $g_2 \cdot g_1$ is a square.

Now, there are many possible candidates for g_2; however, there's a rather nice classification for them all:
Let $s(x)$ denote the square-free part of x.
$g_1 \cdot g_2$ is a square *if and only if* $s(g_1) = s(g_2)$.

Since $A_i \le 10^7$, we're only dealing with GCDs also upto 10^7.
With the help of a sieve, all the $s(x)$ values upto 10^7 can be precomputed.

This allows us to restate our task: with R_1 and g_1 (and hence some range of L_1) fixed, how many subarrays after R_1 are such that $s(g_1) = s(g_2)$?

This can be computed quickly with the help of a sweep line.
Let S be an array, such that $\mathrm{S}[x]$ denotes the sum of lengths of all subarrays with square-free part x.
Initially, this is filled with zeros.
We'll iterate R_1 from N down to 1.
When at R_1:

- Process all subarrays *ending* at R_1, where as noted above there'll be $\mathrm{O}(\log R_1)$ distinct values of g_1, each with a corresponding range of L_1.
  For each range, say of length k, add $k \cdot \mathrm{S}[s(g_1)]$ to the answer.
- Then, process all subarrays *starting* at R_1 to update $\mathrm{S}[s(g_1)]$ appropriately, so that they're available for earlier elements.

This is enough to get AC.
Since there are multiple test cases, S must either be maintained as a global array (that is reset appropriately for

each test), or as a map.

TIME COMPLEXITY:

$\mathrm{O}(N \log N \log \max A)$ per testcase.

PROBLEM:

Given N and M, find the number of arrays of length N containing distinct integers between 1 and M, such that if $i < j$, then A_i shouldn't have more distinct prime factors than A_j.

EXPLANATION:

Let's first compute the number of distinct prime factors for each integer between 1 and M.
This can be easily done with a sieve in $\mathrm{O}(M \log \log M)$.

Let $\mathrm{freq}[x]$ be the number of integers with x distinct prime factors.

Suppose we fix which N elements out of $[1, M]$ are going to be placed in A.
Then, since we want $\mathrm{pr}(A_i) \le \mathrm{pr}(A_j)$ whenever $i < j$, the order of these elements is *almost* fixed.
The only leeway we have is when two elements have the same prime count, in which case they can appear in any order.
More generally, we can order the elements with a fixed prime count however we like, but that's also the only freedom we have.

This leads to a somewhat natural dynamic programming solution.
Let $\mathrm{dp}(i, j)$ denote the number of valid arrays of length j such that we've only used elements with prime counts $\le i$.
Then, we have

$$\mathrm{dp}(i, j) = \sum_{k=0}^{\mathrm{freq}[i]} \mathrm{dp}(i-1, j-k) \cdot \binom{\mathrm{freq}[i]}{k} \cdot k!$$

That is,

- Fix k, the number of elements with prime count $= i$ that our array contains.
- Fix which k integers with this prime count we're choosing, and their order: $\binom{\mathrm{freq}[x]}{k} \cdot k!$ possibilities.
- Finally, the remaining array has $j - k$ elements with prime count $< i$, which there are $\mathrm{dp}(i-1, j-k)$ ways to form.

Now, since $M \le 10^5$ any integer has at most 6 distinct prime factors, meaning we're only concerned with $i \le 6$.
This makes our dp have $7N$ states, with an $\mathrm{O}(N)$ transition from each one, for $\mathrm{O}(N^2)$ overall.

To optimize this, observe that the transition structure is somewhat special: it's the sum of a product of terms that are based on $j - k$ and k (meaning they sum up to j).
This should remind you of polynomial multiplication!

That is, if we had the polynomials

$$p(x) = \sum_{k=0}^{N} \mathrm{dp}(i-1, k) x^k$$
$$q(x) = \sum_{k=0}^{N} \left(\binom{\mathrm{freq}[i]}{k} \cdot k!\right) x^k$$

Then the j-th coefficient of their product would be exactly $\mathrm{dp}(i, j)$.

The product of two N-degree polynomials can be computed in $\mathrm{O}(N \log N)$ with the help of the Fast Fourier Transform.
In our case, we want the coefficients modulo 998244353, which is achieved by a slight modification of FFT called the Number Theoretic Transform.
You may read about them here.

So, simply replacing the dp transition computation with NTT speeds up our algorithm to $\mathrm{O}(7N \log N)$, which is fast enough to get AC.

TIME COMPLEXITY

$\mathrm{O}(M \log^2 M)$ per testcase.

PROBLEM:

A certain binary search process on an array (described in the statement) returns the *sum* of all the values it encounters along the way, till it finds the required key or fails to do so.

Answer M queries of the following form:

- Given x, find the sum of answers returned by the binary search, across all permutations of $\{1, 2, \ldots, N\}$.

EXPLANATION:

First, let's try to solve a single query.
Suppose we know x, and we want to find the sum of `binsearch`$(P, 1, N, x)$ across all permutations P of $\{1, 2, \ldots, N\}$.

Consider the index where the binary search process terminates when searching for x, say i.
Notice that the path to i is uniquely determined: it has length $\mathrm{O}(\log N)$, and is obtained by a unique sequence of left/right moves, depending on how P_{mid} compares to x.

In particular, if we place elements that compare correctly at these positions, we can always force the binary search to terminate at i, regardless of what the rest of P looks like.

Suppose we require L elements *strictly* smaller than x, G elements *strictly* larger than x, and E elements equal to x (note that E is always 0 or 1, depending on how the search terminates).
Let's count the contribution of each integer from 1 to N to this configuration.

Consider some $1 \le y < x$. In how many permutations does y occur in the path to i when searching for x?
A bit of combinatorics tells us that:

- y can be placed in any of the L positions for "less than" elements; for L ways.
- For the remaining $L - 1$ positions, we can choose and place them in $\binom{x-2}{L-1} \cdot (L-1)!$ ways.
- The "greater than" positions are similarly filled in $\binom{N-x}{G} \cdot G!$ ways.
- If $E = 1$, x must be placed at index i; otherwise we don't really care.
- Finally, all the remaining elements can be permuted in $(N - L - G - E)!$ ways.

So, the overall contribution of y is

$$y \cdot L\binom{x-2}{L-1}\binom{N-x}{G}(L-1)!G!(N-L-G-E)!$$

The multiplier is the same for all y, so after computing it once, it can be multiplied by the sum of all $y < x$ to get their overall contribution in $\mathrm{O}(1)$.

Similarly, the contribution of all $y > x$ can be computed in $\mathrm{O}(1)$ time.
Also, if $E = 1$ then $P_i = x$ must hold, so compute the contribution of x itself in similar fashion.

Notice that we now have a solution in $\mathrm{O}(N)$ time for a single query: fix each index to be the terminating point of the binary search, and compute the contribution in $\mathrm{O}(1)$ time as we did above.

To optimize this further, observe that the actual index i didn't matter at all: we only cared about L, G, and E: the results of the comparisons on the way to index i.
In particular, if multiple different indices had the same (L, G, E) triplet, they'd all have the same contribution to the answer.

Further, note that there can only be $\mathrm{O}(\log^2 N)$ distinct such triplets at all, since $0 \le L + G + E \le \lceil \log_2 N \rceil$ and $0 \le E \le 1$.

So, we can first simulate a single binary search on the entire array, to find all possible (L, G, E) triplets and their counts.
Then, for each query, compute the contribution of each triplet multiplied by its frequency, and sum them all up to obtain the final answer.

TIME COMPLEXITY

$\mathrm{O}(N \log N + M \log^2 N)$ per testcase.

PROBLEM:

For a fixed array A, define

$$f_{\text{and}}(A,k) = \sum_{\substack{S \subseteq A \\ |S|=k}} \text{AND}(S)$$

Similarly define $f_{\text{or}}(A,k)$ and $f_{\text{xor}}(A,k)$.

You're given the values of $f_{op_1}(A,k) \pmod p$ for all $1 \le k \le N$.
Given another bitwise operation op_2, find $f_{op_2}(A,k) \pmod p$ for all $1 \le k \le N$.

EXPLANATION:

Let's first figure out how to transform bitwise AND to bitwise OR (and vice versa), which is subtask 1.

The key behind solving this task is to look at the given operations as *set* operations: bitwise AND is intersection, and bitwise OR is union.
Each A_i can be thought of as the set of bits representing it.

Relating unions and intersections is, of course, the inclusion-exclusion principle.
For instance, the most basic version of it, for two sets, states that

$$|A \cup B| = |A| + |B| - |A \cap B|$$

and more generally, for a collection of sets $S_1, S_2, \ldots, S_n$ we have

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{i=1}^{n} |S_i| - \sum_{1 \le i < j \le n} |S_i \cap S_j| + \sum_{1 \le i < j < k \le n} |S_i \cap S_j \cap S_k| - \ldots + (-1)^{n-1} |S_1 \cap S_2 \cap \ldots \cap S_n|$$

It's easy to see that if each S_i is a set of numbers, this applies not just to the sizes of the sets, but also to their sums: the sum of the union can be found by alternately adding and subtracting (depending on size parity) the sums of the intersections.

Let's apply this to our problem.
Note that $f_{\text{and}}(A,k)$ is pretty much just the sum of the intersections of the A_i, taken k at a time, and similarly $f_{\text{or}}(A,k)$ is the sum of the union of the A_i, taken k at a time.

So, let's say we want to find $f_{\text{or}}(A,k)$.
Let's fix a subset of k elements.
Then, by the inclusion-exclusion principle as mentioned above, the sum of their union (which is their bitwise OR) is just the alternating sum of intersections of these k elements, taken $1, 2, 3, \ldots, k$ at a time.

Looking at it another way, suppose we fix a subset of size $m \le k$.
Then, the sum of the intersection of this subset appears in the expressions of $\binom{N-m}{k-m}$ subsets' unions (depending on which other elements are picked).
This is independent of the subset itself, so the overall contribution of *all* subsets of size m is just $(-1)^{m-1}\binom{N-m}{k-m} \cdot f_{\text{and}}(A,m)$.

Notice that this already solves the problem!
$f_{\text{or}}(A, k)$ can be computed in $\mathrm{O}(N)$ time, by iterating across all $m \leq k$ and computing the appropriate contribution of size m subsets to the resulting sum.
Simply do this for all k to get a solution in $\mathrm{O}(N^2)$, which is good enough.

Moving from unions to intersections can be done similarly, write out the appropriate inclusion-exclusion expression to find multipliers for each size.

Next, we bring XOR into the picture.
But really, nothing much changes at all!

Bitwise XOR can be thought of as the symmetric difference operation on sets, and once again it's possible to write inclusion-exclusion expressions for it in terms of unions/intersections (and vice versa).

This is not particularly hard to do, by looking at how things work for small sets.
For instance we have (see here for why):

$$\left|\bigoplus_{i=1}^{k} S_i\right| = \sum |S_i| - 2\sum |A_i \cap A_j| + 4\sum |A_i \cap A_j \cap A_k| - \dots$$

Once the appropriate expression is found, the solution is exactly the same as before: for each k, find the contribution of subsets of size $m \leq k$ to it via some simple combinatorics, and sum up all the answers.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ per testcase.

PROBLEM:

N people have M friendships among them. Each person also has a strength A_i.
Friendship is transitive.

Find the number of orders S of these people such that, for any two friends i and j such that $A_i < A_j$, i appears before j in S.

EXPLANATION:

The M friendships define an undirected graph among the N people.
Further, the friendship relation being transitive really just splits the people up based on their connected components: i and j are friends with each other *if and only if* they're in the same connected component.

So, as a first step, find all connected components of the graph.
This can be done in a variety of ways: depth-first search, breadth-first search, or even a DSU.

Now, observe that:

- Two different components don't interfere with each others' orders at all.
- Within a single connected component, the people must be ordered by increasing strength.
  However, for a fixed strength within this component, the people can be ordered in any way.

Let's first find the number of arrangements for a single component.
This is not too hard: as noted above, the only choice we have at all is to move around people with the same strength.
So, if there are $\text{ct}[x]$ people with frequency x in a given component, the number of ways to arrange them is $(\text{ct}[x])!$, the factorial of $\text{ct}[x]$.
So, for a single component, the total number of arrangements is the product of $(\text{ct}[x])!$ across all x that appear in the component.
If the component is known, this is quite easy to compute: build a frequency table of all the elements in the component, then directly find the product of the required factorials.

Next, we need to think about interactions between components.
Suppose there are k components, with sizes $s_1, s_2, \ldots, s_k$.
Suppose we've also fixed an order of elements for each of the components.
How many ways do we have to arrange them into an overall order?

Answer

Recall that there's no interaction between components at all.
So, the only thing that matters is *which* positions are chosen by the different components.

Thinking of it differently, we have k types of balls, and s_i of the i-th type. We'd like to find the number of ways to arrange these balls in a line, where balls of the same type aren't distinguished.
That number is just

$$\frac{N!}{s_1!s_2!\ldots s_k!}$$

One way of seeing this is as follows:

- There are $\binom{N}{s_1}$ ways to choose which s_1 positions the first type occupies.
- There are $\binom{N-s_1}{s_2}$ ways to chose the positions of the second type.

- There are $\binom{N-s_1-s_2}{s_3}$ ways for the third type, and so on till $\binom{s_k}{s_k}$ ways for the k-th type.
  Multiplying out everything and cancelling out factorials from the numerator/denominator will give the above expression.

See also this article, specifically the “Number of unique permutations of words” section.

Note that this problem involves division under modulo.
That cannot be done directly, and must instead be done using the concept of modular inverses.
You can learn about them from this article.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

Given N, find the number of pairs of arrays A and B of length N such that:

- $1 \le A_i, B_i \le N$
- $A_i \neq B_i$
- There exists a way to choose either A_i or B_i for each i such that all chosen values are distinct, i.e, form a permutation.

EXPLANATION:

Our first aim should be to characterize when a pair of arrays is 'good', i.e, when it's possible to choose elements such that the final result is a permutation of $\{1, 2, \ldots, N\}$.

Consider a graph G on N vertices and N edges, where for each i we create an edge between A_i and B_i.
Claim: The pair (A, B) is *good* if and only if every component of G contains a cycle.
Equivalently, every component of G contains as many edges as it does vertices.

Proof

Suppose some connected component of G is a tree.
For each edge, we can choose only one of its endpoints.
This component has k vertices but only $k-1$ edges; so some value will be unchosen no matter what.
This means the final sequence will never be a permutation.

So, every component of G *must* contain a cycle.
For a component with a cycle, it's easy to see that choosing every vertex is possible: pick some vertex u on a cycle, then build a spanning tree of the component rooted at u that excludes one of the cycle edges adjacent to it.
For each edge of the spanning tree, pick the child vertex - everything other than u will be chosen.
Finally, use the previously excluded edge to pick u.

So, instead of counting pairs of arrays, we can instead count graphs that satisfy this structure.

A preliminary idea is as follows:

- Let $f(N)$ denote the answer for N.
- Fix the size of the component containing N, say k.
  Then, there are $\binom{N-1}{k-1}$ ways to choose which elements other than N go into the component; and $\binom{N}{k}$ ways to choose which positions of the array they represent.
  Next, there are $f(N-k)$ ways to create the rest of the elements.
  Finally, let $g(k)$ be the number of ways to arrange the k values we've chosen into a **connected component**.
  Putting everything together, we obtain

$$f(N) = \sum_{k=1}^{N} \binom{N-1}{k-1}\binom{N}{k} \cdot f(N-k) \cdot g(k)$$

So, if we knew all the $g(k)$ values, the problem would be solved in $\mathrm{O}(N^2)$ time easily.
Let's focus on computing the values of g now.

$g(N)$ denotes the number of connected labelled graphs with N nodes and N edges (with a couple extra factors specific to this problem for orderings, which we'll come back to at the end).
With N vertices and N edges, the graph will look like a single cycle with some trees hanging off of it.

Let's fix the size of the cycle, say x.
$A_i = B_i$ is not allowed, meaning self-loops aren't allowed. So, we have $x \geq 2$.
There are $\binom{N}{x}$ ways to choose which vertices go into the cycle.
There are $(x-1)!$ ways to arrange there vertices into a cycle; which is further divided by 2 if $x \geq 3$ because we don't distinguish between a cycle and its reverse.

Finally, we have to arrange all the remaining vertices.
It turns out that there are exactly $x \cdot N^{N-x-1}$ ways to do this.
This formula follows from the fact that we have with us $N - x + 1$ components: one cycle of size k and $N - x$ single vertices; and we want to add edges between them to connect them.
In general, the number of ways of adding edges between connected components of sizes $s_1, s_2, \ldots, s_k$ to connect them is

$$s_1 s_2 \ldots s_k \cdot N^{k-2}$$

A proof of this fact can be found at the bottom of this page.

Finally, there are a couple more things to consider:

- Each 'edge' represents the *unordered* pair (A_i, B_i), but we're counting ordered pairs.
  There are two choices for each edge and N edges; so we get a factor of 2^N.
- Next, when computing $f(N)$ we accounted for choosing the positions we're assigning to the connected component, but not their order.
  Accounting for that gives us an extra $N!$ term.

Putting it together and reducing a couple of terms, we have

$$g(N) = \sum_{x=1}^{N} \binom{N}{x} x! \cdot N^{N-x-1} \cdot N! \cdot 2^{N-1}$$

Once again, it's easy to compute this in $\mathrm{O}(N)$ for a fixed N, and so $\mathrm{O}(N^2)$ for all N.

Once all the $g(N)$ values are known, all the $f(N)$ values can be found in $\mathrm{O}(N)$ as earlier and we're done.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per testcase.

PROBLEM:

Magnus and Hikaru play infinitely many chess games.
Initially, Magnus' rating is R_m and Hikaru's is R_h. A player's winning chance is directly proportional to their rating.
Winning a game increases a player's rating by 1, losing doesn't change it.

Find the probability that Magnus has won exactly P games and Hikaru has won exactly Q.

EXPLANATION:

The fact that winning probability is directly proportional to rating means that, if two players with ratings x and y play against each other, their chances of winning are $\frac{x}{x+y}$ and $\frac{y}{x+y}$, respectively.

For Magnus to win exactly P games and Hikaru to win exactly Q, clearly exactly $P+Q$ games must be played.
So, let's find the probability that, after $P+Q$ games, Magnus has won P.

Suppose we fix exactly which of the P games were won by Magnus: say, games $x_1, x_2, \ldots, x_P$,
The probability of this happening is as follows:

- Hikaru wins everything before the x_1-th game.
  - So, he wins games $1, 2, 3, \ldots, x_1$.
  - For this to happen has a chance of $\frac{R_h}{R_m+R_h} \cdot \frac{R_h+1}{R_m+R_h+1} \cdot \frac{R_h+2}{R_m+R_h+2} \cdot \ldots \cdot \frac{R_h+x_i-2}{R_m+R_h+x_i-2}$
- Then, Magnus wins game x_i, with probability $\frac{R_m}{R_m+R_h+x_i-1}$
- Then, Hikaru wins everything between x_i+1 and x_2-1; once again this can be written out into a product; and so on.

At first glance, this appears to depend on the x_i values chosen - but it actually doesn't!
Let's look at the numerators and denominators of the fractions we're multiplying separately.

- For the i-th game played, the total rating increase of the players before it is exactly $i-1$.
  So, the denominators will always be the sequence $(R_m+R_h), (R_m+R_h+1), (R_m+R_h+2), \ldots, (R_m+R_h+P+Q-1)$.
  This is independent of the x_i values.
  If factorials are precomputed, their product can be found in $\mathrm{O}(1)$ time since it's just $\frac{(R_m+R_h+P+Q-1)!}{(R_m+R_h-1)!}$
- As for the numerators, they also follow a pattern.
  - Let's look at the games won by Magnus.
    For the first one, his rating is R_m. For the second, it's (R_m+1). The third, (R_m+2), and so on — because only when winning does it change.
    So, across all games won by Magnus, the product of ratings in the numerator is simply $R_m \cdot (R_m+1) \cdot (R_m+2) \cdot \ldots (R_m+P-1)$.
    Once again, this can be computed quickly with precomputed factorials.
  - The games won by Hikaru follow a similar pattern, with their product being Q contiguous numbers starting from R_h.

So, we're able to quickly compute the probability that Magnus wins a fixed subset of P matches.
This number is independent of the actual subset, so to find the probability that Magnus wins *some* subset of P

matches, we simply multiply this probability by the number of ways of picking a subset of P matches to win: which is just $\binom{P+Q}{P}$.

So, the final answer is just

$$\binom{P+Q}{Q} \cdot \frac{(R_m+P-1)!}{(R_m-1)!} \cdot \frac{(R_h+Q-1)!}{(R_h-1)!} \cdot \left(\frac{(R_m+R_h+P+Q-1)!}{(R_m+R_h-1)!}\right)^{-1}$$

where the last term is inverted because it belongs to the denominator.
If all factorials and inverse factorials are precomputed, this can be found in $\mathrm{O}(1)$ time.
However, $\mathrm{O}(\log MOD)$ per testcase is also fast enough (i.e, only precomputing factorials).

Note that the largest value we see is $(R_m+R_h+P+Q-1)$ which can be almost $4 \cdot 10^6$, so precomputation needs to be till there.

TIME COMPLEXITY

$\mathrm{O}(1)$ per testcase after precomputation of $4 \cdot 10^6$ factorials and inverse factorials.

PROBLEM:

For a fixed parameter D and a multiset S, the **cost** of a partition of S into several non-empty multisets $S_1, S_2, \ldots, S_k$ equals

$$\sum_{i=1}^{N}(D + \max(S_i) - \min(S_i))$$

Define $F(S, D)$ to be the minimum cost of a partition of S with parameter D.

You're given N, M, D, and K.
Find the number of distinct multisets A of length N such that:

- $1 \le A_i \le M$ for each i; and
- $F(A, D) = K$

EXPLANATION:

First, let's see how to compute $F(A, D)$ for a fixed multiset A.

Computing F(A, D)

Let $A_1 \le A_2 \le \ldots \le A_N$ be the multiset.
We'll call a partition *optimal* if it attains $F(A, D)$.

Claim: There exists an optimal partition such that every subset will consist of a *contiguous segment* of the A_i.
Proof: Consider some partition of A into subsets. Suppose there are indices $i < j < k$ such that i and k belong to the same subset (say S_1), but j doesn't (say it's in S_2).
Then, moving j from S_2 to S_1:

- Doesn't change the contribution of S_1 to the cost at all, since j is in the middle.
- Doesn't *increase* the contribution of S_2 to the cost; since either it was in the middle and didn't matter, or was an endpoint and its removal brought the endpoints closer, hence lowering cost.

So, moving j to S_1 is not worse.
By repeatedly performing this process on the j with minimal index, we see that in at most N moves we reach a state where each subset is a segment, thus proving the claim.

Now, consider some partition of A into segments; say there are k segments $[L_i, R_i]$.
The cost of this partition is

$$\begin{aligned}\sum_{i=1}^{k}(D + A_{R_i} - A_{L_i}) &= k \cdot D + \sum_{i=1}^{k}(A_{R_i} - A_{L_i}) \\ &= k \cdot D + \sum_{i=1}^{k}\left(\sum_{j=L_i}^{R_i-1}(A_{j+1} - A_j)\right)\end{aligned}$$

This means we're essentially summing up differences between all pairs of adjacent elements that lie in the same segment; and adding D for each adjacent pair that *isn't* in the same segment (along with one extra D term).
Thinking of this differently, for each adjacent pair of elements (A_i, A_{i+1}), we can:

- Take them into the same segment, for a cost of $A_{i+1} - A_i$; or
- Keep them in different segments, for a cost of D.

Together, this tells us that the minimum possible cost is simply

$$F(A, D) = D + \sum_{i=2}^{N} \min(A_i - A_{i-1}, D)$$

To apply the above condition, we also need a nice enough model of a multiset.

Consider a multiset A of length N with elements between 1 and M, such that $A_i \le A_{i+1}$.
Note that this multiset is determined uniquely by:

- The value of A_1;
- The sequence of adjacent differences $(A_{i+1} - A_i)$; and
- The value A_N

This is a useful characterization because as we saw earlier, we're interested in adjacent differences.
Further, if we define $A_0 := 0$ and $A_{N+1} := M$, then A_1 and A_N are also defined by adjacent differences (to 0 and M respectively).

That is, if we let $B_i = A_i - A_{i-1}$, then A is determined uniquely by the $N + 1$ values $[B_1, B_2, \ldots, B_{N+1}]$.
Note that there are a couple of constraints on the B_i values:

- $B_1 \ge 1$, because we want $A_1 \ge 1$.
- $B_1 + B_2 + \ldots + B_{N+1} = M$, since we start at 0 and end at M.
  Considering $(B_1 - 1)$ instead of B_1, we replace the above two constraints by $B_1 \ge 0$ and $sum(B_i) = M - 1$ instead.

Let's now see how we can count valid multisets.
B_1 and B_{N+1} don't contribute to the cost at all.
For each $1 < i \le N$, we add $\min(D, B_i)$ to the cost.
There's always an extra D added to the cost; so let's just work with $K' = K - D$ as the target cost instead.

We'll treat elements that are $< D$ and $\ge D$ differently.
Suppose x of the B_i are $\ge D$.
We fix their positions in $C(N - 1, x)$ ways.
Note that:

- Each of these x indices contribute D to the cost, and *must* contain values that are $\ge D$.
- Each of the other $N - 1 - x$ indices contribute B_i to the cost, and *must* contain values that are $< D$.
- Further, the sum of B_i in the second case must equal exactly $K' - x \cdot D$ for the total cost to equal K'.
- B_1 and B_{N+1} are mostly unconstrained; they just need to be ≥ 0, and as noted above the sum of all B_i should be $M - 1$.

Without loss of generality, suppose $B_2, B_3, \ldots, B_{x+1}$ are to be $\ge D$.
Then, for each of them, we can write $B_i = D + C_i$, where $C_i \ge 0$.

The C_i are otherwise unconstrained.

Putting everything together, we have:

$$\begin{gathered} B_1 + B_2 + \ldots + B_{N+1} = M - 1 \\ B_1 + B_{N+1} + (B_2 + B_3 + \ldots + B_{x+1}) + (B_{x+2} + \ldots + B_N) = M - 1 \\ B_1 + B_{N+1} + x \cdot D + (C_2 + C_3 + \ldots + C_{x+1}) + (K^{'} - x \cdot D) = M - 1 \\ B_1 + B_{N+1} + C_2 + C_3 + \ldots + C_{x+1} = M - 1 - K^{'} \end{gathered}$$

Here, each of the C_i, and both B_1 and B_{N+1} only need to be non-negative integers.
The number of solutions to this equation can thus be found by stars and bars.

That only leaves the values $[B_{x+2}, B_{x+3}, \ldots, B_N]$.
Each of these must be $< D$, and their overall sum must equal $K^{'} - x \cdot D$.
Counting the number of solutions to this can be done by combining the stars-and-bars method with inclusion-exclusion.

How?

If there were no upper bound D, this is a direct application of stars-and-bars.

Now, let's define P_i: the number of arrangements such that *exactly* i of the indices violate the upper bound, i.e, are $\geq D$.
We want to compute P_0.
As is usual for inclusion-exclusion tasks, when exactness is hard to deal with, we relax the constraints a bit.
Define Q_i to be the number of arrangements such that *at least* i of the indices violate the upper bound.

Computing Q_i is not hard: fix which of the i positions are being violated, and then everything is free with the target sum being $K^{'} - x \cdot D - i \cdot D$ (using a similar trick as we did earlier when converting B_i to C_i).

Inclusion-exclusion then tells us that

$$P_0 = \sum_{i=0}^{N-1-x} (-1)^i \binom{N-1-x}{i} Q_i$$

which can be computed in $\mathrm{O}(N)$ time once all the Q_i are known.

The problem is thus solved in $\mathrm{O}(N^2)$ time.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per testcase

PROBLEM:

You're given a tree with N vertices, representing N cities and connections between them.
City i needs A_i towers built in it.

Initially, city 1 has a brick factory, and no other cities do. You can perform the following moves:

- Build a brick factory at some city, such that at least one neighbor of this city already has a brick factory.
- Build one tower at a city with a brick factory.

Find the number of ways of building all the towers using the minimum number of moves.

EXPLANATION:

Notice that the condition on building brick factors essentially boils down to:
"Root the tree at vertex 1. Then, a brick factory can be built at vertex u only if a factory has already been built at the parent of u".

Let's deal with a slightly simpler problem first: suppose we didn't have to build any towers, and just wanted to build one brick factory at *every* city (still following the above rule).
How many possible ways are there of doing this?

Answer

This is a somewhat well-known 21 idea, and can be found with the help of dynamic programming.

Let dp_u be the number of ways to perform the process for vertices in the subtree of u.
Let sz_u denote the size of the subtree of u.
Then, for any u whose children are $v_1, v_2, \ldots, v_k$, we have:

- Exactly sz_u moves will be made in this subtree.
  Of them, the first is of course to build a factory at u itself.
- After that, note that the process independently continues to each of the children of u.
  We can perform moves in the subtree of each child, and moves performed within one subtree don't affect moves performed in other subtrees: they can be interspersed as we like.
- So, we can do the following:
  - From the remaining $sz_u - 1$ positions, sz_{v_1} of them go to the first subtree, sz_{v_2} go to the second subtree, and so on.
    There are
    $\binom{sz_u - 1}{sz_{v_1}}\binom{sz_u - 1 - sz_{v_1}}{sz_{v_2}} \ldots \binom{sz_u - 1 - sz_{v_1} - \ldots - sz_{v_{k-1}}}{sz_{v_k}}$
    ways to perform this distribution, which can be further simplified down to
    $\frac{(sz_u - 1)!}{(sz_{v_1})!(sz_{v_2})! \ldots (sz_{v_k})!}$
  - After distributing positions, each subtree has its own orders as well.
    This gives us a further multiplier of $dp_{v_1} \cdot dp_{v_2} \cdot \ldots \cdot dp_{v_k}$.
- With the base condition $dp_u = 1$ for a leaf, this allows us to compute all the dp_u values; and hence find dp_1 which is what we wanted.

We'll adapt this method to our current problem as well.

Let's first find the minimum number of moves needed.
Clearly, we need one move for each tower; so that's $A_1 + A_2 + \ldots + A_N$ moves.

Apart from that, every city with $A_i > 0$ needs a brick factory built in it.
This further means that if $A_u > 0$, then every *ancestor* of u needs a brick factory built.
Turning this around, we see that city u needs a brick factory built if and only if some city in the *subtree* of u requires a tower; and subtree sums can be precomputed for all vertices in $\mathrm{O}(N)$ using a dfs.

Now we need to count the number of orders of performing these moves.
That can be done using subtree dp in a similar fashion as what was described above.
Let $\mathrm{moves}[u]$ denote the number of moves we need for the subtree rooted at u, and $\mathrm{dp}[u]$ denote the number of ways to perform these moves.
Then,

- As noted above, $\mathrm{moves}[u] = A_u + \sum \mathrm{moves}[v]$ across all children v of u.
  Further, if $\mathrm{moves}[u] > 0$, further add 1 to it, because we need to build a brick factory at u first.
- Next, we'll compute $\mathrm{dp}[u]$.
  The logic is pretty much the exact same as what was done for the easier version:
  - The first move is fixed, we build a brick factory at u.
  - After that, moves within its children are independent; and these are also independent moves from building towers at u.
  - So, of the remaining $\mathrm{moves}[u] - 1$ positions:
    - Fix A_u of them to build towers at for city u.
    - For each child v of u, fix $\mathrm{moves}[v]$ positions where the operations on this subtree will occur, and also account for all orders of moves within this subtree itself.
  - Together, this (upon a bit of reduction) gives us:

$$\mathrm{dp}[u] = \frac{(\mathrm{moves}[u] - 1)!}{(A_u)! \cdot \prod(\mathrm{moves}[v])!} \cdot \prod \mathrm{dp}[v]$$

where the products are across all children v of u.

If factorials and inverse factorials are precomputed, this solves the problem in $\mathrm{O}(N)$ time: the final answer is just $\mathrm{dp}[1]$.
Note that $\mathrm{moves}[1]$ can be as large as $10^6 + N$, so make sure to precompute upto an appropriate upper bound (and not just till 10^6).

As an aside, the first 'simpler' problem we solved in fact somewhat directly solves this version too!
Consider a new tree where for each u, we attach a chain of size A_u to it as a subtree.
The answer is then just the number of orderings of this new tree (after pruning useless subtrees), which we already saw how to compute.

Of course, directly implementing this will result in a TLE because it's proportional to the sum of A_i, and that's bounded for a single testcase but not over all testcases.
However, it does explain why the final expressions end up looking remarkably similar.

TIME COMPLEXITY

Precomputation of $2 \cdot 10^6$ values, followed by $\mathrm{O}(N)$ per testcase.

PROBLEM:

Given a string S of length N, find the number of ways of partitioning it into substrings such that each resulting substring has exactly k vowels.

EXPLANATION:

Let pos_i denote the position of the i-th vowel in string S.

Let's look at how we can partition S into substrings containing exactly k vowels.

- The first k vowels must be in the same substring; and this substring cannot include the $(k+1)$-th vowel.
  So, the first 'cut' must be made after pos_k, but before pos_{k+1}.
  There are $\text{pos}_{k+1} - \text{pos}_k$ such choices.
- The next consecutive k vowels must be in the same substring.
  By the same reasoning as above, the second cut must thus be made after pos_{2k}, but before pos_{2k+1}.
  There are $\text{pos}_{2k+1} - \text{pos}_{2k}$ choices here.
- More generally, it's easy to see that for the i-th cut, there are $\text{pos}_{ik+1} - \text{pos}_{ik}$ choices.

Finally, note that each cut is independent of the others.
So, the total number of ways is just the product of all these numbers.

In particular, if the string has $x \cdot k$ vowels, we need exactly $x - 1$ cuts to separate them.
So, the answer will be

$$(\text{pos}_{k+1} - \text{pos}_k) \cdot (\text{pos}_{2k+1} - \text{pos}_{2k}) \cdot \ldots \cdot (\text{pos}_{(x-1)k+1} - \text{pos}_{(x-1)k})$$

The pos_i values can be computed by just iterating across the string, so this algorithm takes $\text{O}(N)$ time.

TIME COMPLEXITY

$\text{O}(N)$ per testcase.

PROBLEM:

Given N, find for each i from 1 to N:

$$\sum_{j=1}^{N} \binom{j-1}{j-\lfloor\frac{j-1}{i}\rfloor-1}$$

EXPLANATION:

There are a couple of different ways to approach this task; one more visual in nature and the other one more algebraic.
They both end up at the same solution, however.

Visual approach

Let's look at what's going on in the context of Pascal's triangle.
For a fixed i:

- Let (x, y) be our position on the triangle. Initially, $(x, y) = (0, 0)$.
- Then, for each j from 1 to N:
  - If j is divisible by i, we move to $(x+1, y)$.
  - Otherwise, we move to $(x+1, y+1)$.

The answer for i is the sum of values at all points we visited.

Looking at the triangle, our path can be broken into several smaller 'straight' paths along diagonals.
In particular, each multiple of i is when we move from one diagonal to another.
So, there are $\lceil\frac{N}{i}\rceil$ such movements in total.
Summing this across all i, we have $\mathrm{O}(N \log N)$ movements in total; which also means $\mathrm{O}(N \log N)$ diagonal segments in total.

Finally, the sum of each diagonal segment can be found in $\mathrm{O}(1)$ time using the hockey-stick identity, and so we're done!

Algebraic approach

Note that $\binom{j-1}{j-\lfloor\frac{j-1}{i}\rfloor-1} = \binom{j-1}{\lfloor\frac{j-1}{i}\rfloor}$, because $\binom{N}{K} = \binom{N}{N-K}$.

Let's fix i. Then, notice that if we also fix $x = \lfloor\frac{j-1}{i}\rfloor$, we obtain a certain range of j that attain this value of x; in particular, $x \cdot i \le j-1 < (x+1) \cdot i$.

Let this range be $[L, R]$. Then, we want to compute $\sum_{k=L}^{R} \binom{k}{x}$.
This can be done using the hockey-stick identity, and equals $\binom{R+1}{x+1} - \binom{L}{x+1}$.

Now, for a fixed i, there are only $\lceil\frac{N}{i}\rceil$ ranges of j we care about; we don't care about any x such that $i \cdot x > N$.

Since the sum of $\frac{N}{i}$ is $\mathrm{O}(N \log N)$, we compute $\mathrm{O}(N \log N)$ binomial coefficients in total; each of which can be done in $\mathrm{O}(1)$ time.
This is enough to solve the problem.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

Given N and M, find the sum of $(\sum_{i=1}^{N} |A_i - B_i|)^2$ across all pairs of arrays A and B of length M such that:

- $0 \le A_i, B_i \le M$
- $sum(A_i) = sum(B_i) = M$

EXPLANATION:

First, let's analyze the value contributed by a single array.
$|A_i - B_i|$ is somewhat hard to deal with, so let's rewrite it as $\max(A_i, B_i) - \min(A_i, B_i)$; after all, that's what their difference is.

Then, notice that

$$\begin{aligned}\sum_{i=1}^{N} |A_i - B_i| &= \sum_{i=1}^{N} (\max(A_i, B_i) - \min(A_i, B_i)) \\ &= \sum_{i=1}^{N} (\max(A_i, B_i) + \min(A_i, B_i)) - 2 \cdot \sum_{i=1}^{N} \min(A_i, B_i) \\ &= 2M - 2 \cdot \sum_{i=1}^{N} \min(A_i, B_i)\end{aligned}$$

So, the sum of differences is fixed if we fix the sum of $\min(A_i, B_i)$ across al indices.
Also, notice that $\sum_{i=1}^{N} \min(A_i, B_i)$ must lie between 0 and M, so there aren't too many possible values for it.

Let's fix $S = \sum_{i=1}^{N} \min(A_i, B_i)$, and try to count how many pairs of arrays A and B attain this value of S.
Each such array will contribute $(2M - 2S)^2$ to the final answer, so if we can quickly count them for a fixed S, we'll be done.

This counting can be done in several steps, as follows:

- First, let's fix the distribution of minimum values across all N indices.
  This is equivalent to saying we have N non-negative integers that sum up to S; and we know there are $\binom{N+S-1}{S}$ such sequences of integers.
- Once this is fixed, we need to distribute a total of $M - S$ more to each of A and B so that their respective sums reach M.
  However, we've already fixed the values of the minimums, so we need to distribute these values only to the maximums; in particular, we can't increase *both* A_i and B_i now, but only at most one of them.
- To account for this, we have the following:
  - Fix the number of positions such that A_i is *strictly* greater than B_i; suppose there are x such positions.
    Note that x can vary from 0 to N.
  - Once x is fixed, also fix which x positions these are: $\binom{N}{x}$ choices in total.
  - After this, we need to distribute $M - S$ to the A_i values of these x positions such that all of them receive at least 1, and we need to distribute $M - S$ to the B_i values of the other $N - x$ positions such all of them receive ≥ 0.

Both of these can be found as pretty much direct applications of stars-and-bars, giving us the product of two binomial coefficients.

Note that after S and x are fixed, we only compute a few binomial coefficients and multiply them out.
There are $M+1$ possible values of S and $N+1$ of x, so this solution is $\mathrm{O}(N \cdot M)$ in total.

tl;dr the answer is

$$\sum_{S=0}^{M} \sum_{x=0}^{N} ((2M-2S)^2 \cdot \binom{N+S-1}{S}\binom{N}{x}\binom{M-S-1}{x-1}\binom{N-x+M-S-1}{M-S})$$

TIME COMPLEXITY

$\mathrm{O}(N \cdot M)$ per test case.

PROBLEM:

There are N asteroids in the 2-D plane, the i-th initially at (X_i, Y_i).
Each second, every asteroid moves one step left.

An optimal strategy to destroy the asteroids is to pick an asteroid with the smallest X-coordinate each second.
A random strategy is to pick a random asteroid each second.

Find the probability that a random strategy is optimal.

EXPLANATION:

Let's number the asteroids $1, 2, \ldots, N$.
Then, any strategy of destroying the asteroids can be described by an arrangement of the numbers 1 to N, i.e, a *permutation* P of length N which tells us that the i-th asteroid destroyed was P_i.

The total number of strategies is thus $N!$, and a random strategy simply picks one out of these $N!$ uniformly at random.
So, if there are K *optimal* strategies in total, the probability that a random strategy is optimal is exactly $\frac{K}{N!}$
This means all we need to do is compute K, the number of optimal strategies.

We can make the following observations:

- The y-coordinates of the asteroids don't matter at all, they neither change nor affect which one is chosen. They can be entirely ignored.
  This makes our problem essentially one-dimensional, we only care about the x-coordinates.
- All the asteroids move similarly, so their relative positions stay the same. That is, if asteroid i is initially to the left of asteroid j, then this relation will continue to hold no matter how much time passes.
  In other words, we can ignore the passage of time entirely, which makes finding an optimal strategy obvious.

Let $x_1, x_2, \ldots, x_k$ be the *distinct* x-coordinates that appear among the points, with $x_1 < x_2 < \ldots < x_k$.
Let c_i be the number of points whose x-coordinate is x_i.

Then, an optimal strategy must look as follows:

- First, pick all c_1 points whose x-coordinate is x_1. These c_1 points can be chosen in any order, for $c_1!$ ways in total.
- Next, pick all c_2 points whose x-coordinate is x_2. These c_2 points can be chosen in any order, for $c_2!$ ways in total.

  $\vdots$
- Finally, pick all c_k points whose x-coordinate is x_k. These c_k points can be chosen in any order, for $c_k!$ ways in total.

The number of optimal strategies is thus simply the product of all the above values, i.e, $\prod_{i=1}^{k} c_i!$

If factorials are precomputed, computing the above product becomes simple.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Given an increasing array A consisting of integers from 1 to N, count the number of permutations of $\{1, 2, \ldots, N\}$ whose prefix maximums are exactly A.

EXPLANATION:

There's a couple of different ways to solve this, but a proper model of what's going on makes it quite simple.

Let's try to create a permutation P by placing the integers $1, 2, \ldots, N$ in order.

First, note that the first element of P *must* be A_1, since it's the first maximum.
So, we have the following:

- 1 can't be placed at the first position, but anything else is ok since it can't affect the prefix maximums. So, there are $N - 1$ options.
- 2 can't be placed at the first position or where 1 is, but again anywhere else is fine. $N - 2$ options.
- Similarly, 3 has $N - 3$ positions where it can be placed.

  $\vdots$
- $A_1 - 1$ has $N - (A_1 - 1)$ choices.

Once these have been placed, we place A_1 at position 1, as noted earlier.

After this, notice that A_2 *must* be placed in the leftmost empty position for it to be the second maximum.
This fixes $A_1 + 1$ elements, so

- Element $A_1 + 1$ has $N - (A_1 + 1)$ options.
- Element $A_1 + 2$ has $N - (A_1 + 2)$ options.

  $\vdots$
- Element $A_2 - 1$ has $N - (A_2 - 1)$ options.

It's easy to see that this process continues till we place $A_K = N$.

That is, for every integer x that's *not* one of the A_i, it has $(N - x)$ options.

So, the final answer is simply the product of all of these, i.e

$$\prod_{\substack{x=1 \\ x \neq A_i}}^{N} (N - x)$$

which is easily computed in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You're given N distinct integers between 1 and $2N$ as an array P.

Consider all permutations Q of $[2N]$ such that $P_i = Q_i$ for $1 \le i \le N$.
Find the number of distinct prefix maximum arrays among all such Q.

EXPLANATION:

tl;dr

The answer is

$$\binom{3N-M}{N} - \binom{3N-M}{N+1}$$

where $M = \max(P_1, P_2, \ldots, P_N)$.

Most of the difficulty of this problem comes from modelling it correctly.
For now, let's ignore the prefix of length N we're given, and just look at prefix maximum arrays as a whole.

Since the prefix maximum array is always going to be sorted, two such arrays will differ if and only if the *counts* of the elements in them differ.
So, it's enough for us to consider counts of elements in the prefix maximum array.

Let c_i denote the number of times i occurs in the prefix maximum array. Then, we have the following constraints:

- $c_i \ge 0$ for every i
- $c_1 + c_2 + \ldots + c_i \le i$ for every i, because the elements $\{1, 2, \ldots, i\}$ together cannot be prefix maximums more than i times.
- $c_1 + c_2 + \ldots + c_{2N} = 2N$.

Any assignment of integers to c_i that satisfies these conditions will give us a unique prefix maximum array, and it's not hard to see that there will always exist a permutation that achieves such an array.

These contraints can be represented as lattice paths!
In particular, we can do the following:

- Start at $(0, 0)$.
- For each $0 \le i < 2N$, make one rightward move followed by c_{i+1} upward moves.

So, after i rightward moves, we would've made exactly $c_1 + c_2 + \ldots + c_i$ upward moves.

Then,

- $c_1 + c_2 + \ldots + c_{2N} = 2N$ ensures that we will always end up at position $(2N, 2N)$.
- $c_1 + c_2 + \ldots + c_i \le i$ means we will always stay below the line $x = y$ (though we can touch it).

It's not hard to see that any path from $(0, 0)$ to $(2N, 2N)$ that always stays below the line $x = y$ similarly allows us to assign values to each c_i, so all we need to do is count the number of such paths!

Counting the number of right-up lattice paths that don't cross the line $x = y$ is a rather well-known problem, and can be done using the reflection principle to subtract 'bad' paths from the total number of paths.

In fact, when starting from $(0, 0)$ and ending at $(2N, 2N)$, the answer is exactly the $2N$-th Catalan number!
A sketch of the proof for this case can be found here.

Let's return to the original problem.
We are given the first N elements of the permutation, which doesn't actually change much: in terms of our path, it simply fixes a prefix of our path.

In particular, if $M = \max(P_1, P_2, \ldots, P_N)$, then the path so far has us end at position (M, N).
So, all we need to do is count the number of paths from (M, N) to $(2N, 2N)$ that stay below the line $x = y$.

By the reflection principle, that works out as follows:

- The total number of paths from (M, N) to $(2N, 2N)$ is $\binom{3N-M}{N}$.
- The 'bad' paths from (M, N) are in bijection with the total number of paths from $(N-1, M+1)$, thus being equal to $\binom{3N-M}{N+1}$

This makes the answer simply

$$\binom{3N-M}{N} - \binom{3N-M}{N+1}$$

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

The beauty of an array B equals $\sum_{i=1}^{M} |B_i - B_{M+1-i}|$.
The score of array B equals the maximum beauty across all its permutations.

Given N and X, find the sum of scores of all arrays of length N whose elements lie between 1 and X.

EXPLANATION:

First, let's figure out what the score of an array is.

Notice that index i and index $M+1-i$ are symmetric about the center of the array, so indices are essentially paired up. The difference of each such pair is counted twice.
Consider some index i, with $B_i = x$ and $B_{M+1-i} = y$.
Then, these indices contribute $2 \cdot |x-y| = 2\max(x,y) - 2\min(x,y)$ to the beauty of the array.

Since indices are paired up, from a global perspective this means that exactly $\lfloor \frac{N}{2} \rfloor$ indices contribute a *positive* value to the beauty, and exactly $\lfloor \frac{N}{2} \rfloor$ indices contribute a *negative* value.

To maximize this, clearly the best we can do is to ensure that the largest $\lfloor \frac{N}{2} \rfloor$ values contribute positively, and the smallest $\lfloor \frac{N}{2} \rfloor$ values contribute negatively.
This is of course easily achievable: simply sort the array!

The above discussion tells us exactly what the score of a fixed array B is — it's twice the sum of the largest $\lfloor \frac{N}{2} \rfloor$ elements, minus twice the sum of the smallest $\lfloor \frac{N}{2} \rfloor$ elements.
Let's use this information to solve the problem.

Suppose we were able to calculate $\text{ct}[i][y]$, the number of arrays whose i-th smallest element is y.
This would allow us to solve the problem quite easily: after all, the answer would either increase or decrease by $2 \cdot y \cdot \text{ct}[i][y]$, depending on the value of i.

However, directly computing $\text{ct}[i][y]$ is not easy.

Instead, let's try to relax the problem a little.
Let $g[i][y]$ be the number of arrays whose i-th smallest element is $\leq y$.
That is, $g[i][y]$ is the number of arrays containing *at least* i elements that are $\leq y$.
Then, $\text{ct}[i][y] = g[i][y] - g[i][y-1]$, so if we're able to compute g we can easily compute ct.

However, once again it's not clear how we would compute g. This time, there's too much freedom: sure, we can place i elements $\leq y$ in the array, but there might be other elements $\leq y$ as well. How to account for those?

Well, we can try adding that as a constraint.
Let $h[i][y]$ denote the number of arrays whose i-th smallest element is $\leq y$, and whose $(i+1)$-th element is $> y$.
In other words, $h[i][y]$ is the number of arrays such that *exactly* i elements are $\leq y$.

$h[i][y]$ is in fact quite easy to compute:

- Fix the positions of the i elements that are $\leq y$. This can be done in $\binom{N}{i}$ ways.
- Each of these i positions has y choices, since we can place any of $1, 2, 3, \ldots, y$ there.
  This is y^i ways.
- Each of the other positions has $X - y$ choices, since we can place any of $y+1, y+2, \ldots, X$ there.
  This is $(X-y)^{N-i}$ ways.

So, $h[i][y] = \binom{N}{i} \cdot y^i \cdot (X-y)^{N-i}$, which can be computed in $\mathrm{O}(\log N)$ or even $\mathrm{O}(1)$ with some precalculation.

Since $h[i][y]$ is the number of arrays with exactly i elements $\leq y$, and $g[i][y]$ is the number of arrays with at least i elements $\leq y$, we have

$$g[i][y] = h[i][y] + h[i+1][y] + \ldots + h[N][y]$$

This is basically a (column-wise) suffix sum of the h array, so is easy to compute once h is known.

So,

- Compute all the $h[i][y]$ values, which can be done in $\mathrm{O}(N \cdot X)$.
- Take column-wise suffix sums of h to obtain the $g[i][y]$ values, which once again takes $\mathrm{O}(N \cdot X)$ time.
- From the $g[i][y]$ values, compute $\mathrm{ct}[i][y]$ and hence solve the problem.

TIME COMPLEXITY

$\mathrm{O}(N \cdot X)$ per test case.

PROBLEM:

A binary string is said to be good if it has an equal number of 01 and 10 substrings.

Given a binary string S, count the number of its good subsequences.

EXPLANATION:

First, we need a simpler characterization as to exactly when a binary string is good.

Let A be a binary string.
Suppose its first character is 0.
Then, A looks like a bunch of zeros, followed by several ones, followed by several zeros, and so on.
Since we only care about adjacent characters when they aren't equal, we can compress the string, i.e, $A = 010101\ldots$.

In this form, it's not hard to see that A has an equal number of 10 and 01 substrings only when it ends with a 0.
This condition is both necessary and sufficient; more importantly, it's quite a simple one, since it depends only on the first and last characters.

This is the characterization we'll use: a binary string A is *good* **if and only if** $A_1 = A_N$.

Given S, our aim is now to count the number of subsequences of S that start and end with the same character.
First off, there are N subsequences of length 1 that always fit this criterion, so let's keep them aside.

Suppose we have $S_i = S_j$, where $i < j$.
Let's count the number of subsequences with these two as the endpoints.

- We can't pick any elements before i or after j.
- Between i and j, we can freely pick any elements.
- There are $(j - i - 1)$ elements between them, so there are 2^{j-i-1} subsequences in total.

This gives us a (slow) solution in $\mathrm{O}(N^2)$ by fixing each pair of (i, j). All that remains is to optimize it.

Let's fix the value of j.
Then, we want to compute the sum of 2^{j-i-1} across all $i < j$ such that $S_i = S_j$.
Notice that $2^{j-i-1} = 2^{j-1} \cdot (2^{-i})$, so all we really want is the sum of 2^{-i} across all $i < j$ such that $S_i = S_j$: we can later multiply this sum by 2^{j-1} to account for *all subsequences ending at index* j.

Maintaining this sum is quite easy!
Computing 2^{-i} for a given i is a standard application of binary exponentiation and modular inverses.
After that, the only thing that needs to be maintained is two separate sums: the sum of 2^{-i} for all indices i such that $S_i = 0$, and the same sum for all indices such that $S_i = 1$.

Then, for a fixed j, the required contribution can be computed in $\mathrm{O}(1)$; after which the corresponding sum can be increased by 2^{-j}.

TIME COMPLEXITY

$\mathrm{O}(N)$ or $\mathrm{O}(N \log MOD)$ per testcase.

PROBLEM:

Given N and M, count the number of sequences A of length N such that:

- $0 \le A_i \le M$
- $|A_i - \text{MEX}(A_1, A_2, \ldots, A_i)| \le 1$

EXPLANATION:

Let's first analyze the structure of an absolute mex sequence.

Suppose we've placed $A_1, A_2, \ldots, A_{i-1}$, and want to decide on A_i.
Let $m_{i-1} = \text{MEX}(A_1, \ldots, A_{i-1})$.
There are now two possibilities: we set $A_i = m_{i-1}$, or we don't.

- If we don't set $A_i = m_{i-1}$, then $\text{MEX}(A_1, \ldots, A_{i-1}, A_i) = m_{i-1}$.
  So, we want $|A_i - m_{i-1}| \le 1$ and $A_i \ne m_{i-1}$, which (for the most part) gives us two choices; namely $m_{i-1} - 1$ and $m_{i-1} + 1$.
- If we set $A_i = m_{i-1}$, then $\text{MEX}(A_1, \ldots, A_{i-1}, A_i) > m_{i-1}$. In particular,
  - If $m_{i-1} + 1$ has occurred in the sequence before this, the new MEX is $m_{i-1} + 2$, which is bad for us since $|A_i - (m_{i+1} - 2)| = 2$.
  - If $m_{i-1} + 1$ hasn't occurred yet, the new MEX is $m_{i-1} + 1$, which is fine.

Together, these observations give us a somewhat rigid structure on what an absolute mex sequence looks like.

- Let's say the sequence goes 'bad' at index i if we place $m_{i-1} + 1$ at it.
- Notice that as soon as a sequence goes bad, every following element must be either $m_{i-1} + 1$ or $m_{i-1} - 1$; since as we noted above, placing m_{i-1} would cause the difference to be 2 which is unacceptable.

So, it's enough to count the following:

- The number of sequences that never go bad; let's call them good sequences
- For each i, the number of sequences that go bad for the first time at index i.

These can be done separately.

Counting good sequences

It's not hard to observe that a good sequence will be of the form

$$[0, 0, 0, \ldots, 0, 1, 1, \ldots, 1, 2, 2, \ldots, 2, 3, 3, \ldots, k]$$

for some $0 \le k \le M$.
That is, some zeros, followed by some ones, followed by some twos, and so on.

Counting the number of such sequences is fairly easy.
Let's fix k, the largest element of the good sequence.
Then, we only need to decide how many times each of $0, 1, 2, \ldots, k$ occur in the sequence, since it'll be sorted.
Also, each of them needs to occur at least once.

This is the classical stars-and-bars problem in combinatorics, and simply equals $\binom{N-1}{k}$.

So, the number of good sequences is simply

$$\sum_{k=0}^{M} \binom{N-1}{k}$$

which can be computed in $\mathrm{O}(\min(N, M))$.
Counting bad sequences

Now, let's count bad sequences.
If the sequence goes bad at $i = 1$, the only possibility is $A = [1, 1, 1, 1, \ldots, 1]$.

For $i \geq 2$, if the sequence goes bad at index i,

- We're placing $m_{i-1} + 1$ at this index; so $m_{i-1} + 1 \leq M$ must hold, i.e, $m_{i-1} \leq M$.
- In particular, indices $1, 2, \ldots, i-1$ must form a good sequence whose mex is m_{i-1}. From above, we know there are $\binom{i-2}{m_{i-1}-1}$ such sequences.
- Positions $i+1, i+2, \ldots, N$ have two choices each: they can be $m_{i-1} - 1$ or $m_{i-1} + 1$. This is 2^{N-i} possibilities.

So, for a fixed i, by considering all possible values of m_{i-1}, we see that the number of sequences is

$$2^{N-i} \times (\binom{i-2}{0} + \binom{i-2}{1} + \ldots + \binom{i-2}{M-2})$$

2^{N-i} is easy to compute, but what about the inner summation?

For that, we can use a small trick.
Suppose we know $(\binom{i-2}{0} + \binom{i-2}{1} + \ldots + \binom{i-2}{M-2})$ for some index i.
When we move to $i+1$, we'd like to know $(\binom{i-1}{0} + \binom{i-1}{1} + \ldots + \binom{i-1}{M-2})$.

Recall Pascal's identity: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Applying this to our summation, we have

$$\begin{aligned}
\sum_{j=0}^{M-2} \binom{i-1}{j} &= \sum_{j=0}^{M-2} (\binom{i-2}{j-1} + \binom{i-2}{j-2}) \\
&= \binom{i-2}{-1} + 2 \times \sum_{j=0}^{M-3} \binom{i-2}{j} + \binom{i-2}{M-2} \\
&= 0 + (2 \times \sum_{j=0}^{M-3} \binom{i-2}{j} + \binom{i-2}{M-2} + \binom{i-2}{M-2}) - \binom{i-2}{M-2} \\
&= 2 \times \sum_{j=0}^{M-2} \binom{i-2}{j} - \binom{i-2}{M-2}
\end{aligned}$$

Notice that we already have $\sum_{j=0}^{M-2} \binom{i-2}{j}$, so we just need to multiply it by 2 and compute $\binom{i-2}{M-2}$ to subtract, and we're done.
This allows to move from i to $i+1$ in $\mathrm{O}(1)$ time, and so we're done.

Note that depending on your implementation, $M = 0$ and $M = 1$ might be special cases (with answers 1 and $N+1$ respectively).

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Given a multiset A, count the number of its subsets whose mode is unique.

EXPLANATION:

For each x, let freq_x denote the frequency of x in the multiset.

Most counting problems require us to fix some property of the object we're counting, and this one is no different. Since we'd like to count the number of subsets with unique mode, the defining factors of a subset are:

- Which element is the mode, and
- The frequency of the mode in this subset

So, let's fix both of these: suppose x is the mode of the subset and occurs k times in this subset; and let's attempt to count the number of valid subsets.

Note that, for any *other* integer y:

- If $\text{freq}_y < k$, then there are $(\text{freq}_y + 1)$ choices for y: it can occur 0 times, 1 time, 2 times, $\ldots$ freq_y times; and all these choices are viable since they don't affect the mode
- If $\text{freq}_y \geq k$, then y has k choices: it can appear anywhere between 0 and $k-1$ times.

These are all independent choices for each y, and so the total number of subsets equals the product of:

- $\prod(\text{freq}_y + 1)$ across all y whose frequency doesn't exceed $k-1$; and
- k^{m-1}, where m is the number of elements whose frequency in A is $\geq k$ (we subtract 1 because x also has frequency $\geq k$ but shouldn't be counted here).

Now, notice that:

- $\prod(\text{freq}_y + 1)$ is pretty much just a prefix product, if frequencies are sorted in ascending order.
- The number of frequencies that are $\geq k$ can also be computed quite easily if the sorted list of frequencies is known (either directly via binary search, or in $\text{O}(1)$ by precomputing some suffix sums).

In particular, we can calculate the number of subsets for the pair (x, k) in $\text{O}(1)$ or $\text{O}(\log N)$ after a bit of precomputation.

This is enough to solve the problem: simply iterate over all pairs of (x, k) (such that $k \geq 1$) and perform this computation; the answer is the sum of all these values.

The number of pairs we consider is exactly $\text{freq}_1 + \text{freq}_2 + \text{freq}_3 + \ldots + \text{freq}_{10^9} = N$, so the time complexity of this is $\text{O}(N \log N)$.

TIME COMPLEXITY

$\text{O}(N \log N)$ per testcase.

PROBLEM:

You have an array A of N integers. It's called K-beautiful if every subarray of size K has the same sum.

You can perform **at most** M moves, each one increasing some A_i by 1.
How many possible K-beautiful final arrays are possible?

EXPLANATION:

Suppose A was a K-beautiful array.
Then, the sum $(A_i + A_{i+1} + \ldots + A_{i+K-1})$ should be constant across all i.
In particular, this means $(A_i + A_{i+1} + \ldots + A_{i+K-1}) = (A_{i+1} + A_{i+2} + \ldots + A_{i+K})$, which tells us that $A_i = A_{i+K}$.

That is, if we fix the first K elements of A, the remaining values are fixed.

First however, we need to make everything equal to a baseline.
For each $1 \le i \le K$, let $M_i = \max(A_i, A_{i+K}, A_{i+2K}, \ldots)$.
We need to first bring all these values up to M_i, which takes $(A_i - M_i) + (A_{i+K} - M_i) + \ldots$ operations.
Compute this for each i and sum up the number of operations needed.

If this baseline number of operations is larger than M, then the answer is immediately 0 since no K-beautiful array can be constructed. Otherwise, remove these operations from M; so now we have upto M 'free' operations.

We are now ready to actually solve the problem.
Note that the final array is entirely determined by the (final) values of $A_1, A_2, \ldots, A_K$.
We know their current values, so the only thing that matters is how much we add to each index.

So, suppose we add x_i to index i.
Each time we add 1 to index i, we also need to add 1 to indices $i + K, i + 2K, i + 3K, \ldots$ in order to preserve the K-beauty condition.
In particular, suppose there are c_i such indices (including i).

The total number of operations we use is then $c_1x_1 + c_2x_2 + \ldots + c_Kx_K$.

Our objective is thus to count the number of solutions to

$$c_1x_1 + c_2x_2 + \ldots + c_Kx_K \le M$$

such that each x_i is ≥ 0.

A quadratic solution

Let's fix m ($0 \le m \le M$) and count the number of solutions to $c_1x_1 + c_2x_2 + \ldots + c_Kx_K = m$.
This should remind you of the classical stars-and-bars problem, but it isn't quite in that form.

First, note that the c_i aren't arbitrary integers: in fact, each one of them is either $\lceil \frac{N}{K} \rceil$ or $\lfloor \frac{N}{K} \rfloor$, depending on i.

For now, let's assume K doesn't divide N so those two numbers are different.
Let $y = \lfloor \frac{N}{K} \rfloor$, and suppose r of the c_i are equal to y.
y and r are constants independent of m.

Rewriting the equation in terms of y and reducing it a bit, we see that

$$c_1x_1 + c_2x_2 + \ldots + c_Kx_K = m$$
$$y \cdot (x_1 + x_2 + \ldots + x_r) + (y+1) \cdot (x_{r+1} + x_{r+2} + \ldots + x_K) = m$$

Now, notice that $(x_1 + x_2 + \ldots + x_r)$ and $(x_{r+1} + x_{r+2} + \ldots + x_K)$ are two essentially independent summations.

Recall that we fixed m. Let's also fix $m_1 = x_1 + x_2 + \ldots + x_r$.
Notice that this uniquely fixes the value of $x_{r+1} + x_{r+2} + \ldots + x_K$, say to m_2.

The equations for m_1 and m_2 are in the stars-and-bars form, and so the number of solutions to each one can be computed in $\mathrm{O}(1)$ time.
The respective coefficients are $\binom{m_1 + r - 1}{m_1}$ and $\binom{m_2 + K - r - 1}{m_2}$.

Simply multiply these two numbers together to obtain the number of solutions for these fixed values of m and m_1.

Iterating over each (m, m_1) pair gives us a solution in $\mathrm{O}(M^2)$, which is too slow.

Note that if K divides N then fixing m also fixes $m_1 = \frac{m}{y}$ (and m_2 doesn't even come into the picture), so this case can be solved in $\mathrm{O}(M)$ time already.

Speeding up

Notice that fixing any two of m, m_1, m_2 fixes the third one uniquely.
So, let's fix just m_1 and see what we get.

If we also fix m_2, the resulting value of m must be $\leq M$.
In particular, m_2 must satisfy the inequality $y \cdot m_1 + (y+1) \cdot m_2 \leq M$.

This means that the valid values of m_2 form a consecutive set of integers starting from 0.
For each one, we need to add $\binom{m_1+r-1}{m_1}\binom{m_2+K-r-1}{m_2}$ to the answer.

Note that the first quantity is fixed since m_1 is fixed, so we just need the sum of $\binom{m_2+K-r-1}{m_2}$ across all valid m_2.
This isn't too hard: since the m_2 form a contiguous range starting from 0, we can create an array V where $V_i = \binom{i+K-r-1}{i}$ and take its prefix sums.

This allows us to process a single m_1 in $\mathrm{O}(1)$ time, so iterating it from 0 to M solves the problem in $\mathrm{O}(N+M)$ time.

TIME COMPLEXITY

$\mathrm{O}(N+M)$ per testcase.

PREREQUISITES:

Point update/range query structures (segment trees/fenwick trees), basic combinatorics

PROBLEM:

There are N rocks, the i-th of which is at position X_i and has weight W_i.

Rocky will eat a subset of these rocks starting from the one with least weight, then moving to the one with second least weight, and so on.
Find the expected distance travelled by Rocky if any subset is equally likely to be chosen.

EXPLANATION:

First off, each subset has an equally likely chance of being chosen, and there are 2^N subsets.
So, to find the expected value, it's enough to compute the total distance traveled by Rocky across all subsets, and then divide this by 2^N.

As is common with several problems of this type (i.e, find the sum of something across all subsets), instead of fixing a subset and computing the desired quantity for it, we can fix a quantity and see how many subsets it occurs in.

In our case, the quantity we're interested in the distance between points, so let's look at that.
In particular, we care about distances between *adjacent* (in terms of weight) points in the chosen subset; since the total distance covered is the sum of these distances.

This quickly gives rise to a solution in $\mathrm{O}(N^2)$, as follows:

- Let's fix two indices i and j (without loss of generality, say $W_i < W_j$), and count the number of subsets in which Rocky travels directly from X_i to X_j.
- In order for this direct travel to happen, the chosen subset cannot contain any weights lying strictly between W_i and W_j.
  On the other hand, any weights $< W_i$ or $> W_j$ can be freely chosen.
- This gives us $2^{W_i-1+N-W_j}$ subsets in which this direct travel happens, each for a distance of $|X_i - X_j|$.

With i and j fixed, this is an $\mathrm{O}(1)$ (or $\mathrm{O}(\log MOD)$) computation, and we have our quadratic solution.

To optimize this, let's fix the value of j, and look at the structure of all i that we need to compute for.

For now, only consider $X_i < X_j$; the other side can be computed similarly.

For a fixed i, the contribution is $(X_j - X_i) \cdot 2^{W_i-1+N-W_j}$.
Here, 2^{N-W_j} is a constant independent of i, so let's ignore it for now and multiply it in the end.
The remaining part can be split into $(X_j \cdot 2^{W_i-1})$ and $-(X_i \cdot 2^{W_i-1})$.

We want the sum of this across all i such that $X_i < X_j$ and $W_i < W_j$.
Notice that all we really need is the sum of all 2^{W_i-1} and the sum of $X_i \cdot 2^{W_i-1}$, which are both quantities depending purely on i.

This looks something like a 2D range query, but it can in fact be done with a simple segment tree.

Let's iterate over values of W_j from 1 to N (note that we're iterating from small to large *weight*, not *index*).
Doing this automatically satisfies the constraint $W_i < W_j$ whenever we do a query, so we only need to take care of the $X_i < X_j$ condition.

This means we can simply query for all indices $< j$ (since the input guarantees that $X_1 < X_2 < \ldots < X_N$).
Knowing the $2^{W_i - 1}$ and $X_i \cdot 2^{W_i - 1}$ values for these indices, this reduces to a simple range query.

In order to ensure that we don't compute these values for indices with weights $> W_j$, set the values at all indices to be zero initially, and update an index's value only *after* it has been processed as a W_j.
This way, when doing the range sum query, the contribution of higher weight indices is 0, which is what we want.

$X_i > X_j$ can be handled similarly, the range query is on a suffix instead of a prefix.
Don't forget to divide by 2^N in the end.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Combinatorics, the inclusion-exclusion principle, polynomial convolution with arbitrary mod

PROBLEM:

A permutation P of length $2N$ is called N-beautiful if the N pairs $(P_i \bmod N, P_{i+N} \bmod N)$ are all distinct.

Given K, For each $1 \le i \le K$ find the number of i-beautiful permutations, modulo a given prime number M.

EXPLANATION:

We'll first solve subtask 1, and then extend that solution to subtask 2.

Subtask 1

The limit on K is small enough here that $\mathrm{O}(K^2)$ will pass.
In particular, we can solve for a fixed N in $\mathrm{O}(N)$ time; so let's attempt to do that.

Note that we only really care about the values of $P_i \bmod N$, which range from 0 to $N-1$. Also, we have exactly two of each value.
We'll call arrays of length $2N$ containing each integer from 0 to $N-1$ twice *good arrays.*

Let's call a permutation that isn't N-beautiful, N-ugly.
We'll try to count the number of N-ugly permutations.

Suppose we have a N-ugly permutation.
Then, there exist two indices $1 \le i < j \le N$ such that $P_i = P_j$ and $P_{i+N} = P_{j+N}$.
Such a pair will be called a *bad pair*.

In particular, note that these four positions use all copies of P_i and P_{i+N}, so the rest of the elements are simply a good array of length $2N-4$ (after appropriate relabelling).

It's not too hard to count the number of such arrays, since fixing i, j, the values P_i and P_{i+N}, and which copy of each value goes to which position is simple math and a bunch of multiplications.

However, we can't directly subtract this count from $(2N)!$, since a good array with multiple bad pairs was counted multiple times!
This should lead you to the idea of using the inclusion-exclusion principle.

Let B_i be the number of good arrays with *at least* i bad pairs.
By the principle of inclusion-exclusion, the answer we're looking for is

$$\sum_{i=0}^{N} (-1)^N B_i$$

So, let's focus on computing B_i quickly.
This is a generalization of how we computed B_1, and is just a bunch of combinatorics.

Details

There are i bad pairs, which fixes $2i$ positions among the first N. These can be chosen in $\binom{N}{2i}$ ways.
We need to now split these $2i$ positions into i pairs.
This can be done in $\binom{2i}{2,2,2,\ldots,2} = \frac{(2i)!}{2^i}$ ways by definition of the multinomial coefficient.

Next, we need to fix which elements go into each pair. Let's fix an order of the pairs and assign values one by one. This can be done in $N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot (N-2i+1)$ ways, which after a little algebraic manipulation is seen to be just $\binom{N}{2i}(2i)!$.

However, we also need to divide by $i!$ since the order in which we assign values to the pairs doesn't actually matter (it makes no difference to the final permutation).

Each bad pair now has four choices (we can swap elements with the same value modulo N to obtain a different permutation with the same bad pair count).
This gives us 4^i choices in total.

Finally, there are $(2N-4i)!$ ways to arrange the elements not among these bad pairs.

Putting everything together, we have

$$B_i = \binom{N}{2i}\binom{N}{2i}\frac{4^i(2i)!(2i)!(2N-4i)!}{2^i i!}$$

which can be computed in $\mathrm{O}(\log N)$ or even $\mathrm{O}(1)$ time, since everything is either a factorial or a power of two (or an inverse of one of them)

Since each B_i can be computed quickly, we can solve for a fixed N in $\mathrm{O}(N)$ time, which is good enough for subtask 1.

Subtask 2

Let's look at our expression for B_i again, and clean it up a bit by cancelling out common terms.
We obtain

$$B_i = \frac{2^i N!N!(2N-4i)!}{(N-2i)!(N-2i)!i!} = \frac{2^i N!N!}{i!}\binom{2N-4i}{N-2i}$$

The $N!$ terms are independent of i, so let's ignore them for now.
Notice that we have the product of an "i-part" comprising $\frac{2^i}{i!}$ and something like an "$N-i$ part" comprising $\binom{2N-4i}{N-2i}$.

This should make you think along the lines of convolution.
However, we can't directly multiply two polynomials: the $N-i$ part doesn't quite depend on $N-i$, it depends on $N-2i$ (and there's no apparent nice way to make the i-part depend on $2i$ instead).
It'd be nice if we had $2N$ instead of N, so let's do just that!

We split the computation into two cases: even N and odd N. Let's deal with the even case first.
So, suppose $N = 2n$.

We now have $\binom{2N-4i}{N-2i} = \binom{4n-4i}{2n-2i} = \binom{4(n-i)}{2(n-i)}$, which does depend on $n-i$.

So, create the polynomials p and q, with coefficients

- $p_i = \frac{(-1)^i 2^i}{i!}$
- $q_i = \binom{4i}{2i}$

and multiply them together so obtain r.

The answer for $2n$ is then r_n, multiplied by $(2n)!^2$ (which we took out at the start).

For odd N, a similar technique works: write $N = 2n + 1$, and once again you'll get a very similar relation on $n - i$. Following this, multiply the appropriate polynomials and we're done.

Note that this problem requires multiplying two polynomials and keeping their coefficients modulo some M, where M isn't necessarily 'nice'.
Standard FFT and NTT won't suffice here; a tutorial on how to do this can be found at cp-algorithms.

TIME COMPLEXITY

$\mathrm{O}(K \log K)$ per testcase.

PROBLEM:

Given two binary strings A and B of length N, count the number of binary strings C of length N such that $\text{dist}(A, C) = \text{dist}(B, C)$.
$\text{dist}(A, C)$ denotes the hamming distance between A and C.

EXPLANATION:

Hamming distance is computed as the sum of N individual terms; one corresponding to each index. So, let's see how each index affects the equality.

Consider an index i.

- If $A_i = B_i$, then this index either contributes 0 to both $\text{dist}(A, C)$ and $\text{dist}(B, C)$ (if $C_i = A_i$); or contributes 1 to both (if $C_i \neq A_i$).
  In other words, it doesn't affect the equality at all, so we can freely choose $C_i = 0$ or $C_i = 1$ here.
- If $A_i \neq B_i$, then depending on our choice of C_i, this index either contributes 1 to $\text{dist}(A, C)$ and 0 to $\text{dist}(B, C)$; or vice versa.
  Clearly, we must choose $\text{dist}(A, C)$ and $\text{dist}(B, C)$ an equal number of times for the equality to hold in the end.

So, suppose there are K positions such that $A_i \neq B_i$; and $N - K$ positions where they're equal.
If K is odd, the answer is immediately 0, since as we noted we need to split these K positions equally. Let's deal with even K now.

At $\frac{K}{2}$ of the unequal positions, we must fix C_i to ensure that $C_i \neq A_i$.
At the other $\frac{K}{2}$ of the unequal positions, we must fix C_i to ensure that $C_i \neq B_i$.
The remaining $N - K$ positions are 'free', and each have two options.

There are $\binom{K}{K/2}$ ways to choose $K/2$ positions out of K, and then the values at these K positions are fixed.
The other positions have 2^{N-K} options in total.
Multiplying everything together, the final answer is

$$\binom{K}{K/2} \times 2^{N-K}$$

Computing binomial coefficients under modulo requires the use of modular division: you can see how here.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PROBLEM:

Two strings X and Y, each of length n, are **similar** if they can be made equal by applying the following operation at most once in each of them.

- Choose any two positions i, j (i can be equal to j) and swap the characters at those positions.

Given a string A of length n, find the number of ordered pairs (X, Y) of *non-similar* permutations of A, modulo $10^9 + 7$.

QUICK EXPLANATION:

An equivalent definition for similar would be: X and Y are similar if you can turn X into Y by performing at most two swaps.

For every lowercase letter c, let A_c be the number of occurrences of c in A. Then the number of permutations of A is $\frac{n!}{A_{"a"}!A_{"b"}!\cdots A_{"z"}!}$. Let this be P.

We count the number of similar pairs of strings instead. The total number of pairs is P^2, so if S is the number of similar pairs, then the number of non-similar pairs is $P^2 - S$.

Let T be the number of strings similar to A. Then it can be shown that $S = PT$. Thus, all we need is to compute T.

Finally, to compute T, we will need to separately count four kinds of similar strings formed from A:

- Strings formed from A with no swaps.
- Strings formed from A with one swap.
- Strings formed from A with two disjoint swaps.
- Strings formed from A with two overlapping swaps.

After computing the A_c values, these counts can be computed in $O(1)$ time each.

EXPLANATION:

String similarity and swap distance

Let's define the **swap distance** between two strings X and Y as the minimum number of swaps needed to turn X into Y. We denote this by $D(X, Y)$. Note the following facts:

- $D(X, X) = 0$
- $D(X, Y) = D(Y, X)$
- $D(X, Y) \leq D(X, Z) + D(Z, Y)$

We can redefine string similarity in terms of swap distance as follows: Two strings X and Y are **similar** if and only if there exists a string Z such that $D(X, Z) \leq 1$ and $D(Z, Y) \leq 1$.

But this also implies that $D(X, Y) \leq D(X, Z) + D(Z, Y) \leq 1 + 1 = 2$. (In other words, it only takes at most 2 swaps to turn X to Y, namely, first turn X to Z, then Z to Y). Thus, one necessary criterion for two strings being similar is that $D(X, Y) \leq 2$. But is this a sufficient criterion? Actually, yes, because if $D(X, Y) \leq 2$, then there must exist some string Z such that $D(X, Z) \leq 1$ and $D(Z, Y) \leq 1$, namely:

- If $D(X, Y) < 2$, then $Z = X$ (or $Z = Y$).
- If $D(X, Y) = 2$, then Z is the *intermediate* string while turning X into Y with two swaps. This Z clearly is one swap distance away from X and Y.

Thus, we now have an equivalent definition for string similarity:

X and Y are similar if and only if $D(X, Y) \leq 2$.

Thus, the problem is equivalent to counting the number of ordered pairs of non-similar permutations of A with swap distance > 2.

Observations

As usual, we'll need a few more observations / insights to make our counting problem easier to solve.

The first observation is that it's easier (or at least it seems to be easier) to count the number of pairs of **similar** permutations instead. This gives us our first observation.

Let P be the number of permutations of string A, and let S be the number of pairs of similar permutations of A. Then the number of pairs of non-similar permutations of A is $P^2 - S$.

To see this, note that P^2 is the number of pairs of permutations of A (because there are P choices for the first and second element of the pair), so if we subtract from it the number of *similar* pairs S, we get $P^2 - S$, the number of *non-similar* pairs.

Thus, we can focus on computing P and S instead. But P is easy to compute using multinomial coefficients [3] :

**For every lowercase letter c, let A_c be the number of occurrences of c in A. Then

$$P = \frac{n!}{A_{\text{"a"}}!A_{\text{"b"}}! \cdots A_{\text{"z"}}!}.$$

**

(There is a possible source of confusion here, namely the usage of lowercase letters to denote variables and the actual characters themselves. To avoid ambiguity, we will always use *quotes* whenever we refer to the actual character. Without quotes, it always denotes a variable. Thus, the c in A_c is a variable, while the "c" in $A_{\text{"c"}}$ is the actual character "c".)

Thus, all that remains is to compute S.

Pairs of similar strings

We need to count the number of similar strings X and Y. Let's fix the string X. Note that X has length n, and we know how many of each letter it has (from the A_c values). We are interested in finding the number of distinct strings Y that can be obtained by performing at most two swaps in X. We will need to distinguish four kinds of strings Y:

- Strings formed from X using no swaps.
- Strings formed from X using one swap.
- Strings formed from X using two overlapping swaps.
- Strings formed from X using two disjoint swaps.

In this classification, we say "swap" to mean "swap between two different letters" to avoid double counting. Since every kind of similar string is exactly one of the above kinds, we can simply count each kind separately and add the

results together.

The first kind is actually very trivial, because there is only one string of that kind, namely X itself. So we'll start on the second kind:

Strings formed from X using one swap.

How many strings Y can be formed from X using exactly one swap of different letters? Let's count it. First, we need to select which letters to swap, say i and j. To avoid double counting, we say $i < j$. Next, we need to select which among the A_i and A_j occurrences of i and j, respectively, will be swapped. There are $A_i \cdot A_j$ such choices. By summing across all (i, j) pairs, we now have our answer!

$$\sum_{\text{"a"} \le i < j \le \text{"z"}} A_i A_j$$

Strings formed from X using two overlapping swaps.

Here, by overlapping swaps, we mean, for example, swapping the i th and j th character, and then the i th and k th character after that. (Notice that two swaps happened in the i th position.) Note that we can safely ignore swaps where $j = k$ because that has the same effect as doing no swaps at all.

Let's understand what precisely happens during two overlapping swaps. Let's say the i th position contains the character c_i, the j th position contains c_j, and the k th position contains c_k. After the first swap, the i th position contains c_j, the j th position contains c_i, and the k th position contains c_k. After the second swap, the i th position contains c_k, the j th position contains c_i, and the k th position contains c_j. Thus, it turns out that two overlapping swaps is equivalent to a single 3-cycle 3 !

So it turns out we are actually counting the number of strings Y that can be obtained from X by doing a 3-cycle on three distinct characters. Let's count it. First, we need to select the three distinct letters to 3-cycle, say i, j and k. To avoid double counting, we say $i < j < k$. Next, we need to select which among the A_i, A_j and A_k occurrences of i, j and k, respectively, will be permuted. There are $A_i A_j A_k$ such choices. Finally, we need to figure out the direction of the three cycle. There are two possible directions: $i \to j \to k \to i$ and $i \to k \to j \to i$. Thus, by summing across all (i, j, k) triples, we get our answer:

$$\sum_{\text{"a"} \le i < j < k \le \text{"z"}} A_i A_j A_k \cdot 2$$

Strings formed from X using two disjoint swaps.

Finally, we now try to count the number of strings that can be formed from X using two *disjoint* swaps. This is actually the most complicated type, because even though the positions of the swaps themselves are distinct, the letters they're swapping might not be. For example, In the word `aabc`, swapping the first and third and then the second and fourth positions results in `bcaa`, and these two swaps are disjoint. However, the letter a was swapped in both swaps!

In fact, we need to classify this type into three subtypes. Let's count them!

Type 1: The characters in both swaps are all distinct.

In this case, we need to select the four characters involved, say i, j, k and l. To avoid double counting, let's say $i < j < k < l$. Next, we need to select which among the occurrences of i, j, k and l will be swapped. There are $A_i A_j A_k A_l$ such choices. Finally, we need to choose which letter swaps with which. There are three possibilities: $(i,j), (k,l)$, or $(i,k), (j,l)$, or $(i,l), (j,k)$. Thus, summing across all choices for (i,j,k,l), we get:

$$\sum_{\text{"a"} \le i < j < k < l \le \text{"z"}} A_i A_j A_k A_l \cdot 3$$

Type 2: The two swaps share one character in common.

In this case, we need to select the three characters involved, say i, j and k. This time, one of the characters will be different from the others, namely the character that will be shared in both swaps. Let's say this character is i. Now, to avoid double counting, we need to let $j < k$. However, i's only restriction is that it must not be equal to j or k (because it is special). Finally, we need to select which among the occurrences of i, j and k will be swapped. There are $A_i(A_i - 1)A_j A_k$ such choices. Note the factor $A_i(A_i - 1)$, because there are A_i ways to choose the place that j will swap with i, and there are $A_i - 1$ ways to choose the place that k will swap with i (which must be different from the previously chosen position). Thus, summing across all choices for (i,j,k), we get:

$$\sum_{\text{"a"} \le i \le \text{"z"}} \sum_{\substack{\text{"a"} \le j < k \le \text{"z"} \\ i \ne j \\ i \ne k}} A_i(A_i - 1)A_j A_k$$

Type 3: The two swaps share two characters in common.

In this case, we need to select the two characters involved, say i and j. (No one is special this time.) To avoid double counting, let's say $i < j$. Finally, we need to select which among the occurrences of i, j will be swapped. There are $\binom{A_i}{2}\binom{A_j}{2}$ such choices (because we need to choose two positions for each letter). Note that this time, it doesn't matter which occurrence swaps with which because the end result will be the same. Thus, summing across all choices for (i,j), we get:

$$\sum_{\text{"a"} \le i < j \le \text{"z"}} \binom{A_i}{2}\binom{A_j}{2}$$

Combining everything

By adding all sums above, we now have computed the number of strings that are similar to X! Let's denote this number by T.

The cool thing about what we did is that we didn't use any property specific to any X other than that it was a permutation of A. Therefore, the sum T is the same for every permutation X of A. Since there are P possible values of X (remember P from above?), it means that S, the number of ordered pairs of similar permutations of A, is simply $P \times T$!

We now all have the necessary things to compute $P^2 - S$, so we now have a working algorithm for the problem! Now, what about the time complexity? The first part is computing the A_is which take $O(n)$ time. Also, P itself takes $O(n)$ time to calculate because of the factorials. Finally, $S = PT$, and all calculations for T depend only on the size of the array $[A_{\text{"a"}}, \ldots, A_{\text{"z"}}]$. But the size of this array is fixed at 26! Therefore, computing T takes $O(1)$ time! The overall algorithm runs in $O(n)$ time.

Don't forget to reduce everything modulo $10^9 + 7$!

The following is an implementation in Python. You can see here a technique of calculating factorials and inverse factorials modulo $10^9 + 7$.

```
from collections import Counter

# compute factorials and inverse factorials
mod = 10**9 + 7
N = 10**5 + 11
fac = [1]*N
ifc = [1]*N
for i in xrange(2,N):
    ifc[i] = (mod - mod/i) * ifc[mod%i] % mod

for i in xrange(2,N):
    fac[i] = fac[i-1] *     i  % mod
    ifc[i] = ifc[i-1] * ifc[i] % mod

for cas in xrange(input()):
    # only the frequencies of letters matter
    A = Counter(raw_input().strip()).values()
    t = 0

    # no swap
    t += 1

    # 1 swap
    for i in xrange(len(A)):
        for j in xrange(i):
            t += A[i] * A[j]

    # 2 swaps, ab cd
    for i in xrange(len(A)):
        for j in xrange(i):
```

Note that Python integers are automatically arbitrary-precision, so we don't need to worry about reducing `t` modulo $10^9 + 7$ every time. However, in many other languages, you may need to do it to avoid **overflow**.

As a final note, the huge amount of nested loops in the code above can be optimized to a single loop using an extensive amount of cumulative sums and dynamic programming. See the editorialist's solution for one way of doing it.

Time Complexity:

$O(n)$

PROBLEM:

For two binary strings x and y of length N, define $F(x, y)$ as follows:

- For a permutation P of $\{1, 2, \ldots, N\}$, let C and D be defined as:
  - $C_i = x_{P_i}$
  - $D_i = y_{P_i}$
- $F(x, y)$ is the maximum value of $\min(\text{LNDS}(C), \text{LNDS}(D))$ across all permutations P.

Let S denote the set of all binary strings of length N. Compute

$$\sum_{x \in S} \sum_{y \in S} F(x, y)$$

EXPLANATION:

Let's first figure out how to quickly compute $F(x, y)$ for a fixed x and y.
Notice that the values x_i and y_i are essentially 'tied' to each other, i.e, need to be moved to the same final position.
This gives us 4 types of positions, depending on which of x_i and y_i are $0/1$.

Let c_{00} denote the number of positions such that $x_i = y_i = 0$. I'll call these 00-positions.
Similarly define c_{01}, c_{10}, c_{11}.
We need to figure out how to arrange these 4 types optimally to maximize the length of the minimum LNDS.

- It's always optimal to place 00-positions at the start of the array, since they'll contribute to the LNDS of both strings.
- For the same reason, it's optimal to place 11-positions at the end of the array.
  - Together, they ensure the minimum LNDS has length at least $c_{00} + c_{11}$.
- Now we have to deal with the 01- and 10-positions.
  Playing around a little with them, it can be seen that the best we can get is $\max(c_{01}, c_{10}) + \frac{\min(c_{01}, c_{10})}{2}$ (where the division is floor division).

Proof

Our claim is that if we have x 01-positions and y 10-positions, the longest possible minimum LNDS has length $\max(x, y) + \min(x, y)2$.

This will be proved in two parts: a lower bound and an upper bound.

Lower bound

First, let's prove the lower bound by constructing a sequence that has this as its minimum LNDS.
w.l.o.g let $x \geq y$, so $\max(x, y) = x$ and $\min(x, y) = y$.

- First place $y/2$ occurrences of 10.
- Then place all x occurrences of 01.
- Finally, place all remaining occurrences of 10.

So, the first string looks like $\underbrace{111\ldots11}_{y/2}\ \underbrace{00\ldots00}_{x}\ \underbrace{11\ldots11}_{y-y/2}$ and the second looks like

$\underbrace{000\ldots00}_{y/2}\ \underbrace{11\ldots11}_{x}\ \underbrace{00\ldots00}_{y-y/2}$

The first $y/2+x$ characters of the first string are non-decreasing, while the last $x+y/2$ characters of the second string are non-decreasing.
This gives us the required lower bound: the minimum LNDS has length at least $x+y/2$.
Upper bound

Now for the upper bound.
Consider a configuration where the first binary string has LNDS of length $> x+y/2$.
In particular, this means that the last $y/2+1$ characters of this LNDS must be all be 1; and everything after them must be 0.
Further, we can also assume that there are no zeros between the ones (if there were, they don't contribute to the LNDS anyway; so we can just move them to the end and it helps out the second string more).

Suppose there are z zeros after these $y/2+1$ positions.
The LNDS of the first string then has length at most $x+y-z$, since the last z positions can't contribute.

Now let's look at the LNDS of the second string.

- If the LNDS doesn't include *any* of these $y/2+1$ positions, then it'll have length strictly less than $x+y/2$ (because there are less than $x+y/2$ remaining positions).
- If the LNDS does include one of them, it might as well include them all: our setup guarantees that the values at these positions are 0 in the second string, and there are no ones inbetween them.
- In particular, the LNDS has length *at most* $y+z$ (where z is as defined above).

We'd like $y+z \geq x+y/2$, which gives us $z \geq x-y/2 \geq y/2$.
In particular, this means the LNDS of the first string has length $\leq x+y-y/2 \leq x+y/2$, a contradiction to us assuming it was strictly greater.

So, the minimum LNDS length cannot exceed $x+y/2$, and our upper bound is proved as well.

With this information in hand, let's compute the answer.
Notice that the score of a pair of strings depends really on only two things:

- How many pairs of same/different positions there are.
- Among the different positions, how many of them are 01 and how many are 10.

Since $N \leq 2000$, we can fix both.
Suppose there are k positions such that $x_i = y_i$; and m of the remaining $N-k$ positions are 01.
Then,

- The LNDS length of such a pair is $k+\max(x,y)+\min(x,y)/2$, where $x=m$ and $y=N-k-m$. Let this be len.
- The number of such pairs can be computed as follows:
  - Fix the set of k equal positions, which can be done in $\binom{N}{k}$ ways.
  - For each equal position, fix whether it's 00 or 11. This is 2^k choices.
  - Of the remaining $N-k$ positions, choose which m of them are 01-positions. This is $\binom{N-k}{m}$ choices.
- Multiply all three values above to obtain the number of pairs of strings, say count.
- Then, the answer increases by $\text{len} \times \text{count}$, since count pairs of strings have an answer of len.

Do this for all k from 0 to N and m from 0 to $N-k$ to obtain the final answer.

For a fixed k and m, we need to compute a couple of binomial coefficients and a power of 2.
The former can be done by precomputing factorials/inverse factorials (or even precomputing all 2000^2 relevant binomials using Pascal's formula), while the latter can be done using binary exponentiation.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ or $\mathrm{O}(N^2 \log MOD)$ per testcase.

PROBLEM:

An array B is called good if B_i is either the maximum or the minimum of $B[1 \dots i]$ for each i.

Given an array A, count the number of its rearrangements that are good.

EXPLANATION:

First, let's sort A since it won't change the answer. From now on, $A_i \le A_{i+1}$.

Let B be a good rearrangement of A. Let's analyze the structure of B.

Let $S_L = [B_{i_1}, B_{i_2}, \dots, B_{i_r}]$ be the subsequence of elements that are prefix minimums, and $S_H = [B_{j_1}, B_{j_2}, \dots, B_{j_s}]$ be the subsequence of elements that are prefix maximums. If an element can be both (for example, B_1) then we treat it as a minimum.
S_L and S_H partition B, so $r + s = N$.

For example, if $B = [2, 3, 3, 1, 4, 1]$ then $S_L = [2, 1, 1]$ and $S_H = [3, 3, 4]$.

Note that the following conditions must hold:

- $B_{i_1} \ge B_{i_2} \ge \dots \ge B_{i_r}$ (the minimums decrease)
- $B_{j_1} \le B_{j_2} \le \dots \le B_{j_s}$ (the maximums increase)
- $B_{j_1} \ge B_{i_1}$

Notice that this is an ordering that encompasses every element!
In particular, this means that S_L contains the smallest r elements of A, and S_H contains the largest s elements of A.

So, let's try to fix the value of r and see how many rearrangements exist such that S_L contains the smallest r elements.

Of course, the first element must be A_r. After that, there are two cases:

- If $A_{r+1} > A_r$ (or $r = N$), we can choose any $r - 1$ positions out of the remaining $N - 1$ and place the elements $A_1, A_2, \dots, A_{r-1}$ in these positions (in descending order) and the other elements in the remaining positions (in ascending order). So, there are $\binom{N-1}{r-1}$ ways for this to happen.
- If $A_{r+1} = A_r$, we need to be a bit more careful. We cannot arbitrarily choose $r - 1$ positions to place the first $r - 1$ elements, because we need to ensure that A_{r+1} is *strictly* the maximum when we place it. This means we need to place at least one element less than A_{r+1} before placing it.
  So, let $x < r$ be the largest index such that $A_x < A_r$. We first place the elements $A_r, A_{r-1}, \dots, A_x$ in the first $r - x + 1$ positions, then we can choose $x - 1$ positions from the remaining $N - (r - x + 1)$ to place the other elements, giving us $\binom{N-r+x-1}{x-1}$ possible ways.
  x can be maintained as you iterate through the array: update it whenever you move from one value to the next.

Adding this up across all positions gives the final answer.

Each binomial coefficient needs to be computed in $\mathrm{O}(\log MOD)$ or $\mathrm{O}(1)$. If you don't know how to do this, read through this article.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Observations, Math, Combinatorics

PROBLEM:

Given a checkered field of size $10^5 \times 10^5$. In this field N different cells (numbered 1 through N) are checkered; for each valid i, the coordinates of the i-th checkered cell are (x_i, y_i). Find minimum possible size of set of cells such that all cells are checkered and each cell is reachable from other cell with minimum possible moves. Also find number of sets with this smallest size.

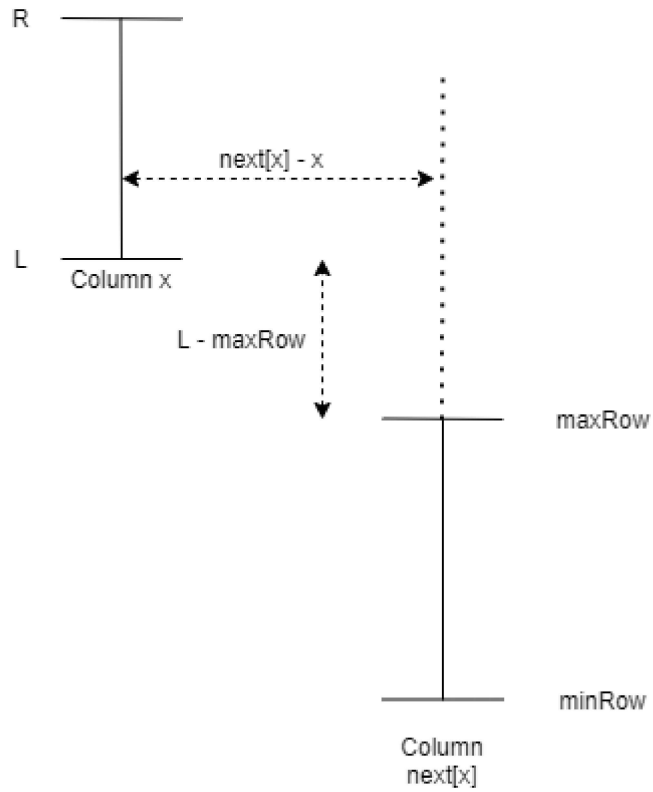
QUICK EXPLANATION

For every column, find minimum length segment such that it covers all checkered cells of the column. Also find next column which has at least one checkered cell. Now start with the first column, find how many cells need to be checkered to move to the next column. Also calculate number of possible ways to move to the next column. This can be done with maintaining minimum segment for the current column and current set of rows to be marked.

EXPLANATION

For any column, find minimum and maximum row number which are checkered. Lets say they are L and R respectively. Then total number of cells which should be checkered for this column would be $R - L + 1$ because we need to connect row L to row R so that they are in same set ie. they are connected through a path with checkered cells.
Lets say we are currently at column x and current segment of checkered cells for this column is $[L, R]$. And next column is $next[x]$. For all other columns $j >= next[x]$, consider all existing checkered cells. Lets say $minRow$ is the minimum row number which has at least one checkered cell and $maxRow$ is the maximum row number which has at least one checkered cell. So now there are three possible scenarios:

Case 1: $maxRow < L$



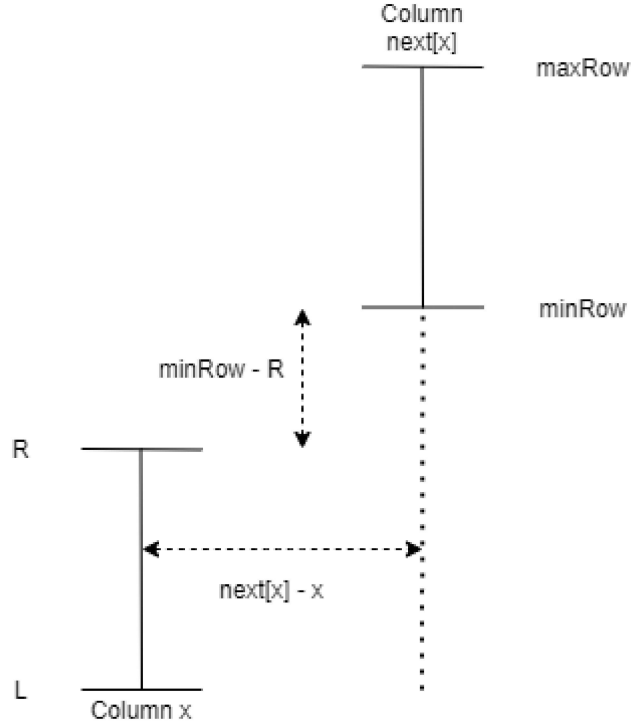
In this case, number of ways to move from column x to $next[x]$ such that all marked cells are in same set with given properties are $\binom{next[x]-x+L-maxRow}{next[x]-x}$.

How

Consider that you are at cell (x_1, y_1) and want to move to cell (x_2, y_2) in a grid. What is total number of paths? Its $\binom{|x_2-x_1|+|y_2-y_1|}{|x_2-x_1|}$. The above problem can also be considered as moving from point (x, L) to point $(next[x], maxRow)$.

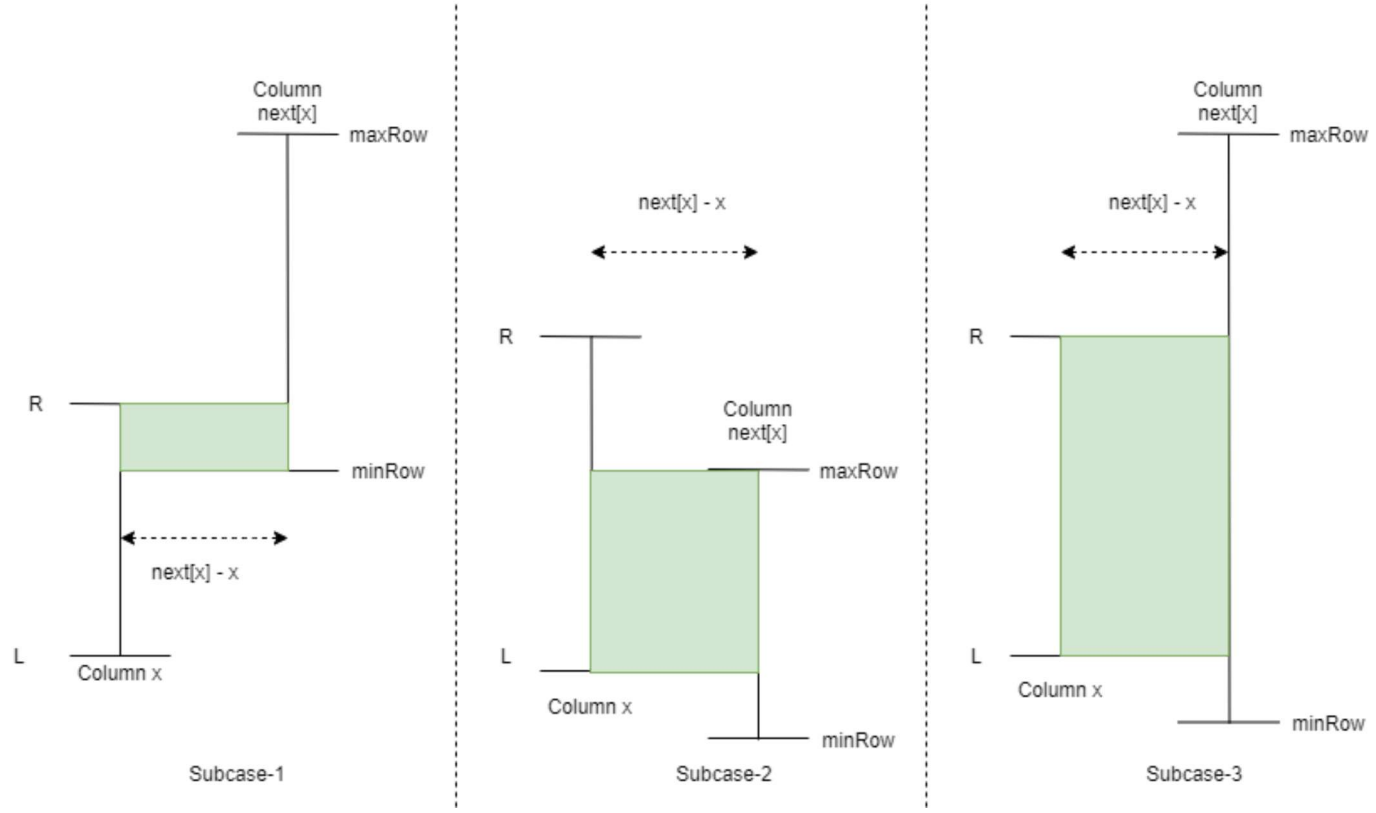
Here number of cells which need to be marked to move from column x to column $next[x]$ is simply $next[x] - x - 1$ because consider column x and column $next[x]$ already marked.
Here when we move to column $next[x]$, L, R would change to,
$L' = L_{next[x]}$ and $R' = L$
Because in column $next[x]$, all cells from $L_{next[x]}$ to L need to be checkered.

Case 2: $minRow > R$



Similar to above case, here number of ways to move from column x to $next[x]$ such that all marked cells are in same set with given properties are $\binom{next[x]-x+minRow-R}{next[x]-x}$.
Here number of cells which need to be marked to move from column x to column $next[x]$ is also $next[x]-x-1$ as in case-1.
Here when we move to column $next[x]$, L, R would change to,
$L^{'} = R$ and $R^{'} = R_{next[x]}$
Because in column $next[x]$, all cells from R to $R_{next[x]}$ need to be checkered.

Case 3: When $[L, R]$ overlaps with $[minRow, maxRow]$



Notice that we have three subcases. In each subcase, the dark area is the region which need to be checkered. In this case, there is only one possible way to move from column x to column $next[x]$ because all the cells in dark area need to be checkered unlike other two cases. In this case number of cells which needs to be checkered to move from column x to $next[x]$ is $(min(R, maxRow) - max(L, minRow) + 1) \times (next[x] - x - 1)$.
Here when we move to column $next[x]$, L, R would change to,
$L^{'} = min(max(L, minRow), L_{next[x]})$
$R^{'} = max(min(R, maxRow), R_{next[x]})$

In the above mentioned solution, for every column, you will need to maintain next column which have at least one checkered cell. This can be precalculated in $\mathrm{O}(N)$ time.
You also need to maintain minimum segment $[L, R]$ for each column which also can be precalculated in $\mathrm{O}(N)$ time.
You also need to maintain current set of rows which has at least one checkered cell to calculate $minRow$ and $maxRow$ as mentioned in the solution. Once we calculated the number of ways and the number of cells to be checkered, we need to remove the current column rows. These two tasks of maintaining the current set of rows and removing the rows belonging to current column can be done with a multiset in $\mathrm{O}(NlogN)$ time.

Complexity Analysis
Overall time complexity of this solution is $\mathrm{O}(NlogN)$.
Space complexity is $\mathrm{O}(N)$ to maintain next columns, minimum range segment, current set of rows and rows belonging to each column.

PROBLEM:

You're given a large number N. Perform the following operation K times and print the length of the resulting number:

- Write N as the minimal sum of powers of 10 (for example, $123 = 100 + 10 + 10 + 1 + 1 + 1$). Then, concatenate these powers in order to form the new value of N

EXPLANATION:

If $K = 0$ the answer is just the length of N.

Otherwise, let's perform the operation on N once and then decrease K by 1.
Notice that the resulting number now contains only the digits 0 and 1; in fact, any further number we create will also consist of only 0's and 1's.
Also notice that the number we created has at most $9 \log_{10} N$ ones, since this is the maximum number of powers of 10 we can extract.

Let's see what happens when we perform the operation on such a number.

Suppose $N = 10^{k_1-1} + 10^{k_2-1} + \ldots + 10^{k_r-1}$ where $k_1 > k_2 > \ldots > k_r \geq 1$.
Note that the length of N is currently k_1, and in particular k_i is the *length* of 10^{k_i-1}.

Now perform the operation once. We'll again get exactly r powers of 10, let their lengths be $l_1 > l_2 > \ldots > l_r \geq 1$.
Working out the details, you can see that:

- $l_r = k_r$
- $l_{r-1} = k_r + k_{r-1}$
- $l_{r-2} = k_r + k_{r-1} + k_{r-2}$

$\vdots$

More generally, $l_i = \sum_{j=i}^{r} k_j$.

Notice that we're just taking the suffix sum of the array $[k_1, k_2, \ldots, k_r]$ to obtain l.
Also note that the length of the resulting number is l_1.

This tells us a way to get to the final answer: compute the array $[k_1, \ldots, k_r]$ from the N we have, then take its suffix sum K times (recall that we decreased K by 1 earlier), and finally print the first element of this suffix sum.

However, K is quite large so directly computing the suffix sum K times is not possible.

Subtask 1

One way of quickly computing suffix sums multiple times is to use matrix exponentiation.

This is an application of a more general technique to quickly compute linear recurrences, a quick tutorial can be found here [3] .

In the case of suffix sums of an array A, we want to iteratively compute $suf_i = A_i + A_{i+1} + \ldots + A_r$.
The result is linear in terms of the values of A, so it's possible to find a matrix that encapsulates this information; in particular the $r \times r$ matrix M with $M_{i,j} = 1$ if and only if $i \leq j$ is what we want.

Note that multiplying M with $[A_1, A_2, \ldots, A_r]^T$ gives us exactly $[suf_1, suf_2, \ldots, suf_r]^T$. Then we can multiply M with the result, and so on K times.
So, the answer we want is to compute the product of M^K with $[A_1, A_2, \ldots, A_r]^T$ and then look at its first element.

Computing M^K can be done in $\mathrm{O}(r^3 \log K)$ time using matrix exponentiation: use binary exponentiation as you would to compute a^b, except the multiplication is now matrix multiplication.

In the first subtask, $N \leq 1000$ which gives us $r \leq 9 \cdot 4 = 40$, so this works in about $40^3 \log K$ operations per test, which is fast enough.

Subtask 2

Matrix exponentiation no longer works here since the number can be quite large: we can have r (the length after one operation) upto $9 \cdot 10^4$ which is too much for $\mathrm{O}(r^3 \log K)$ to work.

Instead, let's use the fact that we don't actually care about the entire suffix sum: we only want its first element after K iterations.

From the fact that we had a linear recurrence, we know that there must exist some coefficients $c_1, c_2, \ldots, c_r$ such that the final answer is $\sum_{i=1}^{r} c_i \cdot A_i$.
Let's try to find these coefficients.

If you try writing out what these coefficients are for a few small values of K, you'll notice something rather interesting: they're nothing but binomial coefficients!
In fact, the coefficients (when rotated 45°) essentially just form Pascal's triangle.
Some intuition for why this happens and related discussion can be found here 8 .

At any rate, we just need to compute some binomial coefficients; in particular, you'll get something like $c_i = \binom{K+i-2}{i-1}$.

The numerator of each coefficient is quite large, so standard methods of computing them won't work.
The denominator is $< r$, and this allows us to compute a single coefficient in $\mathrm{O}(r)$, but this is still a $\mathrm{O}(r^2)$ solution overall, which is too slow.

Instead, let's use the fact that the binomial coefficients are of a special type.
Suppose we knew $\binom{K+i-2}{i-1}$. Can we get $\binom{K+(i+1)-2}{(i+1)-1} = \binom{K+i-1}{i}$ from it?

Yes we can!

This allows us to move from one coefficient to the other in $\mathrm{O}(\log MOD)$ (for division, though you can also do $\mathrm{O}(1)$ by precomputing all inverses).
In particular, slower languages such as Python might require precomputing of inverses.

So, we have a solution in $\mathrm{O}(r)$ or $\mathrm{O}(r \log MOD)$, which is good enough since $r \leq 9 \cdot 10^4$.

TIME COMPLEXITY:

$\mathrm{O}(\log N)$ per testcase.

PREREQUISITES: Basic Combinatorics and Mathematics.

The problem in short:

Given an NxN grid which can consist of either +1 or -1 in each cell. You need to find the number of grids such that the product of elements in exactly one row and one column is -1 and all other rows' and columns' products must be +1. This grid is called to be a Magical grid.

Quick explanation:

Let us consider a 2x2 grid, currently initialized with zero see Fig 1.1. Let's say I want to make the product of row 1 and column 1 to -1, for now, let's consider the [2, 2]th element be +1 see Fig 1.2. Now to make the product of row 2 and column 2 to +1, the elements [1, 2] and [2, 1] must be +1 see Fig 1.3. Now In order to make the product of row 1 and column 1 to -1, the [1,1]th element must be -1 see Fig 1.4. We found a required grid!

0 0	0 0	0 1	-1 1
0 0	0 1	1 1	1 1
(Fig 1.1)	(Fig 1.2)	(Fig 1.3)	(Fig 1.4)

This is the basic idea of how we would solve for larger value for N but using combinatorics.

Full explanation:

Let us consider a 3x3 grid, currently initialized with zero see Fig 2.1. Let's select a row and a column whose row product and column product needs to be -1. Since it is a is 3x3 grid, there are 3 different ways to select rows as well 3 different ways to select columns, so there are 3x3 (or N^2) different ways to select different rows and columns.

0 0 0
0 0 0
0 0 0
(Fig 2.1)

Let's say we selected row 1 and column 1 whose row product and column product will be -1. Now just for some time let's neglect the selected row and column and focus on the elements which would make the other rows' product and column's products +1, which would be the grid $(3-1)^2$ or $(N-1)^2$ see Fig 2.2.

0 0 0
0 0 0
0 0 0
(Fig 2.2)

Now we need to fill this 2x2 grid (for N=3) randomly. Note that there are 2 ways to fill each element in this grid i.e, either -1 or +1 and there are 4 elements need to be filled, so the number of different ways these four elements can be filled is 2^4 ways or $2^{(3-1)^2}$ ways or $2^{(N-1)^2}$ ways, for example, see Fig 2.3.

0	0	0
0	1	-1
0	-1	1

(Fig 2.3)

The reason why we randomly fill these elements is that when you do this the elements of the selected row and column would be bound to have -1 or +1 such that the selected row and column would have the row product and column product as -1 and all other rows and columns would have the row and column product as +1. Thus resulting us with the desired grid, see Fig 2.4.

-1	-1	-1
-1	1	-1
-1	-1	1

(Fig 2.4)

Finally by the fundamental principle of counting:

The number of Magical grids for any given N would be $N^2 * 2^{(N-1)^2}$
In this case for N=3,
Number of Magical grids = $3^2 * 2^{(3-1)^2}$
= 9 * 24
= 9 * 16
= 144.

Hope you understood.

PROBLEM:

Alice picks a random permutation P of size N and a random position i between 1 and N.
Then, as long as $i < N$ and $P_{i+1} < P_i$, she moves to $i + 1$.
What is the expected number of positions she will visit?

EXPLANATION:

There are $N \cdot N!$ possible starting states: choose a permutation, then choose a position.

For each $1 \le i \le N$, let S_i denote the number of starting states that result in Alice visiting exactly i positions.
Then, by definition of expected value, the answer is

$$\frac{1}{N \cdot N!} \sum_{i=1} i \cdot S_i$$

Now let's look at how to compute S_i.
Suppose we fix i. What is needed to visit exactly i positions?

Well, suppose we also fix our starting position, say k. Then, we must have

- $k + i - 1 = N$ and $P_k > P_{k+1} > \ldots > P_{k+i-1}$; or
- $k + i - 1 < N$ and $P_k > P_{k+1} > \ldots > P_{k+i-1} < P_{k+i}$

Let's count each case separately, since they're independent.

Case 1

Consider the case when $k + i - 1 = N$.
Let's fix what the last i elements are. This can be done in $\binom{N}{i}$ ways.
Now there is exactly one way to arrange them, since they must be in descending order.
Further, the other $N - i$ elements can be arranged in $(N - i)!$ ways.

This gives us a total of $\binom{N}{i}(N - i)!$ ways.

Case 2

Consider the case when $k + i - 1 < N$.
First, let's fix the starting position k: there are $N - i$ choices for it (it can be anything between 1 and $N - i$).

With the starting position fixed, suppose we fixed which i elements are in these i positions. However, how do we ensure that the next element is greater than the minimum of these i?

Simple: we simply fix all the $i + 1$ elements instead!
That is, choose $i + 1$ elements in $\binom{N}{i+1}$ ways. Then, choose which one of them is the last element: there are i choices, since we cannot choose the minimum of the chosen $i + 1$ but anything else is ok.
Now, there is only one way to arrange the remaining i elements: descending order.

Finally, the unchosen $(N - i - 1)$ elements can be arranged in $(N - i - 1)!$ ways.

So, the number of ways in this case is $(N - i) \cdot \binom{N}{i+1} \cdot i \cdot (N - i - 1)! = \binom{N}{i+1} \cdot i \cdot (N - i)!$.

Putting these together, we find

$$S_i = (N-i)! \times (\binom{N}{i} + \binom{N}{i+1} \cdot i)$$

Each term here is either a factorial or a binomial coefficient. As linked in the prerequisites section, they can all be computed in $\mathrm{O}(1)$ if factorials are precomputed.

So, each S_i can be computed in $\mathrm{O}(1)$ time. Do this, then find $\sum S_i \cdot i$ and finally divide it by $N \cdot N!$ to obtain the answer.

Note that divisions need be performed under modulo, i.e, finding the multiplicative inverse and multiplying by it.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Frequency arrays, Basic combinatorics

PROBLEM:

You are given an array A and an integer K. Count the number of subsequences that don't contain any pair whose sum is divisible by K.

EXPLANATION:

First, notice that the condition "$A_i + A_j$ is divisible by K" can be written as $A_i + A_j \equiv 0 \pmod{K}$.

In particular, we can work with all the array elements modulo K so that they're all between 0 and $K-1$.

Now, consider what happens when we have $x + y \equiv 0 \pmod{K}$ when both x and y are less than K. There are three possibilities:

- First, we can have $x = y = 0$
- Second, if K is even we can have $x = y = K/2$
- Finally, if neither of the above hold, we must have $y = K - x$; and in particular $x \neq y$.

Let's leave the first two cases alone for now, and look at the third.
For convenience, let $x < K - x$.
Note that for each x, any good subsequence can have *either* some occurrences of x, *or* some occurrences of $K - x$: never both.
In particular, we can take as many x-s as we like, or as many $(K-x)$-s as we like, without affecting any other sums (since $K - (K - x) = x$). Essentially, we 'pair up' x with $K - x$, and then different pairs are completely independent.

So, the choices of which of the x's or $(K-x)$'s we take are completely independent across different x.
This means that any subsequence can be constructed as follows:

- Choose a subset of 1's *or* a subset of $K-1$'s
- Then, choose a subset of 2's *or* a subset of $(K-2)$'s
- Then, choose a subset of 3's *or* a subset of $(K-3)$'s

$\vdots$

Thus, the total number of subsequences can be found by multiplying the number of choices for different x.

This brings us to the questions: how many choices are there for a fixed x?

Answer

Let $freq(x)$ be the number of occurrences of x in the array.

Note that we can choose any subset of the x's, or any subset of the $(K-x)$'s.
The first one gives us $2^{freq(x)}$ choices, while the second gives us $2^{freq(K-x)}$ choices.

The empty set is counted in both, so we need to subtract 1 to avoid overcounting.
This brings the total to $2^{freq(x)} + 2^{freq(K-x)} - 1$.

The number of subsequences is thus just the product of $(2^{freq(x)} + 2^{freq(K-x)} - 1)$ across all x such that $x < K - x$.

The only exceptions here are $x = 0$ and (if K is even) $x = K/2$, which shouldn't be included in the above product because they behave slightly differently. Do you see how to deal with them?

Answer

$x = 0$ and $x = K/2$ follow a simple rule: there can't be more than one of each in the subsequence.

So, we have $1 + freq(0)$ choices for 0 (choose none of them, or choose exactly one), and similarly $1 + freq(K/2)$ choices for $K/2$.

Multiply these quantities to the previous value to obtain the final answer.

Notice that the value for a given x requires us to compute a power of 2 modulo something.
There are several ways to do this: the simplest is to just precompute the value of $2^x \pmod{MOD}$ for every $0 \le x \le 5 \cdot 10^5$ before processing any test cases, after which these can be used in $\mathrm{O}(1)$.
Alternately, you can use binary exponentiation.

TIME COMPLEXITY

$\mathrm{O}(N + K)$ per test case.

PROBLEM:

Given N, x, y, find the number of arrays P satisfying the following:

- P contains exactly x ones, y twos, and $N - x - y$ threes
- $P_1 = P_N = 1$
- No two adjacent elements of P are equal.

EXPLANATION:

This is a pretty much a purely combinatorial task.

The main observation is as follows:
Consider two adjacent ones in any valid array P. Then, the subarray between them must consist of alternating twos and threes.
In particular, if we fix the first element of this subarray, that fixes the entire subarray.

Now, note that any subarray between two ones will be in one of four forms:

- Even length, and starting with either 2 or 3
- Odd length, and starting with either 2 or 3

The even length arrays are functionally equivalent since they contain an equal number of twos and threes.
However, the odd subarrays aren't equivalent: the starting element appears one more time than the rest. Let's try to use this.

Suppose there are k_1 odd length subarrays that start with a 2, k_2 odd length subarrays that start with a 3, and k_3 even length subarrays. Then, we have the following equations:

- $k_1 + k_2 + k_3 = x - 1$, since every 1 except the last has one of these subarrays immediately following it
- $k_1 - k_2 = y - z$, since each odd-length subarray starting with 2 contributes one extra 2, and each odd-length subarray starting with 3 contributes one extra 3.

Let's fix the value of k_1 (it can be anything from 0 to $x - 1$).
Note that the above equations allow us to compute uniquely the values of k_2 and k_3.
If either k_2 or k_3 are invalid (i.e, negative or $> x - 1$), then ignore this value of k_1.

So, suppose we know k_1, k_2, k_3. Let's try to count the number of arrays with these parameters.

That can be done in a few steps.

- First, there are $x - 1$ subarrays. Let's fix which ones of them are even length and which are odd: this gives us $\binom{x-1}{k_3}$ choices.
- Among the odd-length ones, let's fix which ones start with 2 and which ones start with 3: this gives us $\binom{k_1+k_2}{k_1}$ choices.
- The even-length subarrays can be arranged in two ways (starting with 2 or 3), so we have 2^{k_3} choices there.
- Finally, we need to count the number of ways to choose lengths for these subarrays. That is an application of the stars-and-bars technique.

How?

There are exactly $x - 1$ subarrays. Suppose the i-th of them has length a_i.

The total length of the subarrays must equal the number of twos and threes, i.e, y+z.
So,

$$a_1 + a_2 + \ldots + a_{x-1} = y + z$$

However, there are some more constraints:

- Exactly $k_1 + k_2$ of these values are odd. Note that we have already fixed which ones are odd with a binomial coefficient, so without loss of generality we can assume $a_1, a_2, \ldots, a_{k_1+k_2}$ are odd and the rest are even.
- The even values must be strictly positive (since if they had length 0, two ones would be adjacent).

Let's write the odd values as $a_i = 2b_i + 1$, and the even values as $a_i = 2 + 2b_i$. Plugging this into the first equation,

$$\begin{aligned}(2b_1 + 1) + \ldots + (2b_{k_1+k_2} + 1) + (2 + 2b_{k_1+k_2+1}) + \ldots + (2 + 2b_{x-1}) &= y + z \\ \implies 2b_1 + 2b_2 + \ldots + 2b_{x-1} &= y + z - (k_1 + k_2) - 2k_3 \\ \implies b_1 + b_2 + \ldots + b_{x-1} &= \frac{y + z - (k_1 + k_2) - 2k_3}{2}\end{aligned}$$

The b_i have no constraint, other than the fact that they have to be non-negative integers.
Any solution to this equation gives us a valid solution to the original equation by reversing the $a_i \to b_i$ process, so it's enough to count the number of solutions to this equation.
Do note that if $y + z - k_1 - k_2 - 2k_3$ is odd, then the equation has no solution.

Counting the number of solutions here is a direct application of stars and bars, and is simply $\binom{M+x-2}{M}$, where $M = \frac{y+z-(k_1+k_2)-2k_3}{2}$.

So, the final solution is to simply add up

$$\binom{x-1}{k_3}\binom{k_1+k_2}{k_1}\binom{M+x-2}{M}2^{k_3}$$

where $M = \frac{y+z-(k_1+k_2)-2k_3}{2}$, across all possible k_1.

Each binomial coefficient can be computed in $\mathrm{O}(1)$ if factorials and their inverses are precomputed, and the power of 2 can be computed in $\mathrm{O}(\log N)$ using binary exponentiation (or all required powers of 2 can be precomputed, since we only need powers $\leq N$ anyway).

This gives us a solution in $\mathrm{O}(N \log N)$.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

You have an $N \times M$ grid, each cell of which can be colored either red or green. The score of a colored grid is the number of right-down paths from $(1, 1)$ to (N, M) such that the number of red cells equals the number of green cells on this path.

Find the sum of scores across possible colorings of the grid.

EXPLANATION:

This task can be solved with a trick that's rather common in counting problems, sometimes called the *contribution trick*.

Whenever you need to fix a configuration and count the number of objects in it that satisfy some property, you can often instead fix an object that satisfies that property and count how many configurations it appears in — the sum of both values is the same.

Let's do the same thing here. We want to

- Fix a coloring of the grid
- Look at a right-down path from $(1, 1)$ to (N, M)
- Count it if it has an equal number of red and green cells

Instead, we will

- Fix a right-down path from $(1, 1)$ to (N, M)
- Color it in such a way that the number of red and green cells is equal
- Then, count the number of grids that contain this path

Summing this across all paths is the the final answer.

Now, let's see how to compute this.

- First, we need to fix a path. There are $\binom{N+M-2}{M-1}$ paths from $(1, 1)$ to (N, M) — this is a well-known result.
  - An easy proof is that you make exactly $N + M - 2$ steps in total with $M - 1$ of them being right and $N - 1$ being down, so fixing the positions of the right steps fixes the path.
- Next, we need to color the path correctly.
  - Let the length of the path be $L. Note that $L = N + M - 1$.
  - If L is odd, there is no way to color the path correctly
  - Otherwise, we simply fix the positions of the green cells in $\binom{L}{L/2}$ ways, which then also fixes the red cells
- Finally, we want to know how many grids this path appears in. We've fixed the colors of cells on the path, but everything else is completely free and has two choices each (red or green).
  - This gives us $2^{N \cdot M - L}$ grids that contain this path.

So, the final answer is simply

$$\binom{N+M-2}{M-1}\binom{L}{L/2}2^{N \cdot M - L}$$

where $L = N + M - 1$. Note that the answer is 0 when L is odd.

The binomial coefficients can be precomputed in a 2000×2000 table using Pascal's identity since we only need something like $\binom{2000}{1000}$ at worst. Similarly, the powers of 2 upto 10^6 can be precomputed.

This allows each query to be answered in $\mathrm{O}(1)$ time.

There are of course other ways to compute binomial coefficients (for example, the standard factorial + inverse factorial method) and powers of 2 (via binary exponentiation), if you want to do something else.

TIME COMPLEXITY

$\mathrm{O}(1)$ or $\mathrm{O}(\log MOD)$ per test case, depending on implementation.

PROBLEM:

The rating of a number is defined to be the number of trailing zeros in the product of its digits (and 1 if this product is 0).

Given two large numbers A and B, find the sum of the ratings of all numbers from $A+1$ to B.

EXPLANATION:

A and B are rather large numbers: they can have upto 1000 digits each, which definitely doesn't allow for them to be read as integers in C++.

A lot of the time, when dealing with large numbers and you need to calculate some function for all numbers $\leq n$, the solution tends to be digit DP. You can read an introduction to the technique here 23 . Some familiarity with the technique will be assumed below.

In our case, let's define the function $f(N)$ to be the sum of the ratings of all numbers from 1 to N. Then, our answer is $f(B)-f(A)$, so we just need to be compute $f(N)$ at a given argument N fast enough.

Note that the rating of a number depends only on two things:

- The number of twos and fives present as factors of its digits.
- Whether the number contains 0 or not. If it does, the rating is always going to be 1.

In particular, for a number that doesn't contain a zero, its rating is simply the minimum of the number of factors of 2 and 5 present among its digits.
For now, let's consider only numbers that don't contain 0 — the ones that do can be taken care of separately later, and it isn't hard to do so.

There's a fairly simple function that comes to mind immediately:
(1-based indexing is used below)
Let $g(i,x,y)$ denote the answer if we are going to place the i-th digit of the number, and there are x powers of 2 and y powers of 5 so far.
Transitions are simple: iterate over which digit d from 1 to 9 is going to be placed here while ensuring that you don't exceed N (recall that we are ignoring 0), calculate the new values of x and y from d (call them x' and y' respectively), and then recurse to $g(i+1,x',y')$ and add it to the answer.
The base case is of course $g(L+1,x,y)=\min(x,y)$ where L is the length of the string N, and the final answer is $g(1,0,0)$.

By memoizing the values of $g(i,x,y)$ (with perhaps extra boolean states to denote whether the current inequality with N is tight or not, and whether the current prefix is non-zero), this solution runs in $\mathrm{O}(L^3\cdot 10)$.
However, $L=10^3$ for us, so this is too slow.

To speed it up, we use a somewhat common DP optimization: reduce the number of states using existing information.
In particular, it is unnecessary to know both x and y — we only really need to know whether there are more 2-s or more 5-s, and how many more there are. This information can be kept with a boolean and an integer, so we've reduced the number of states by a factor of L.

However, making this change also necessitates a change in the definition of our g, since the parameters are different now.

It's somewhat hard to explain properly with words, and I recommend looking at the setter's code linked below for what to do. However, the gist is as follows:

- $g(state)$ now stores a pair: the answer for that state, and the number of ways to reach that state. Let's denote these by $ans(state)$ and $ways(state)$ respectively.
- If the current state has more 2-s than 5-s, and the current digit adds x more 2-s (where $x > 0$), nothing much needs to be done: the difference between the number of twos and fives grows by x.
- A similar condition applies to the current state having more 5-s and the current digit being 5
- Now, suppose the state has more 2-s and we add a 5. Then,
  - The difference decreases by 1 (unless it was already zero)
  - Let $state_2$ be the state we recurse into. Then, we need to further add $ways(state_2)$ to $ans(state)$ to account for this extra five.
- Something similar is to be done for the case when currently there are more 5-s and the current digit contains a 2. In particular, $\{2, 6\}, \{4\}, \{8\}$ all need to be treated as separate cases since they add $1/2/3$ twos respectively.
- $ways(state)$ is the sum of $ways(state_2)$ across all states that are recursed into.

This brings us down to something like $\mathrm{O}(L^2 \cdot 8)$ states, and 10 transitions from each, which is fast enough for $L = 1000$.

Now, all that remains is to count the contribution of numbers that contain a zero.
Note that this is essentially the same as counting the numbers $\leq N$ that contain a zero, which is not very hard to do.
For example, you can do the following:

- For each i from 1 to L, suppose the first time you go below N is at position i. So, the first $i - 1$ digits match those of N, and the last $L - i$ digits are completely free.
- Suppose you place digit d at this position.
  - If $d = 0$ or the first $i - 1$ digits contain a zero, the remaining digits can be absolutely anything. So, add 10^{L-i} to the answer
  - Otherwise, you need to count the number of strings of length $L - i$ that contain a zero. This can be precomputed with another dp.

Special care must be taken when placing $d = 0$ at $i = 1$, since the above discussion only applies when the number has a non-zero prefix.
However, placing 0 at $i = 1$ is the same as counting all numbers of length $\leq N - 1$ that contain a zero, which can again be precomputed.

Note that this is not the only way to solve the problem: I noticed that many contestants (and even the setters) have many different solutions, some perhaps easier than the one outlined here.
If you would like to share your solution in the comments below, feel free to do so!

TIME COMPLEXITY

$\mathrm{O}(L^2 \cdot 10)$ per test case, with a constant factor of around 8.

PROBLEM:

JJ has an array A of length N, where each element lies between 0 and $2^K - 1$. In one move, he can pick two integers x and i such that $0 \le x < 2^K$ and $1 \le i \le N$, and:

- Set $A_j := A_j \ \& \ x$ for the prefix ending at i, or
- Set $A_j := A_j \mid x$ for the suffix starting at i

How many different arrays can he create by performing this operation several times?

EXPLANATION:

First, note that the problem can be solved for each bit independently since there is no limit on the number of operations. So, if we are able to solve the problem for $K = 1$, we can apply this solution to every bit separately and multiply all the answers together to obtain the final answer.

Now, we just need to solve the problem for $K = 1$, i.e, the array contains only 0-s and 1-s. When this is the case, our operations reduce to:

- Setting some prefix of the array to 0 (the first operation, with $x = 0$)
- Setting some suffix of the array to 1 (the second operation, with $x = 1$)

Note that performing the first operation with $x = 1$ or the second operation with $x = 0$ don't change the array at all, so we can safely ignore them.

This gives us the following observations:

- It is enough to perform at most one operation of each type
- The prefix chosen for the first operation and the suffix chosen for the second can be chosen in such a way that they don't intersect.
- If an operation is performed on the prefix of length i, and $A_i = 0$ initially, we can achieve the same result by performing the operation on the prefix of length $i - 1$ instead. So, it is enough to consider the prefix operation only for those i such that $A_i = 1$.
- Similarly, it is enough to consider the suffix operation only for those i such that $A_i = 0$.

Putting all this together, we can see that the problem simply reduces to counting the number of 10-subsequences of A (do you see why?), which can be done easily in $\mathrm{O}(N)$.

Don't forget to account for the case where no prefix operation and/or no suffix operation is performed!

As mentioned at the start, the final answer is obtained by solving this subproblem for each of the K bits, and multiplying all the answers together.

TIME COMPLEXITY:

$\mathrm{O}(N \cdot K)$ per test case.

PROBLEM:

For an array A of length N, let $F(A)$ denote the sum of the product of all the subarrays of A. Formally,

$$F(A) = \sum_{L=1}^{N} \sum_{R=L}^{N} (\prod_{i=L}^{R} A_i)$$

For example, let $A = [1, 0, 1]$, then there are 6 possible subarrays:

- Subarray $[1, 1]$ has product $= 1$
- Subarray $[1, 2]$ has product $= 0$
- Subarray $[1, 3]$ has product $= 0$
- Subarray $[2, 2]$ has product $= 0$
- Subarray $[2, 3]$ has product $= 0$
- Subarray $[3, 3]$ has product $= 1$

So $F(A) = 1 + 1 = 2$.

Given a **binary** array A, determine the sum of $F(A)$ over all the $N!$ orderings of A modulo 998244353.

Note that orderings here are defined in terms of indices, not elements; which is why every array of length N has $N!$ orderings. For example, the $3! = 6$ orderings of $A = [1, 0, 1]$ are:

- $[1, 0, 1]$ corresponding to indices $[1, 2, 3]$
- $[1, 1, 0]$ corresponding to indices $[1, 3, 2]$
- $[0, 1, 1]$ corresponding to indices $[2, 1, 3]$
- $[0, 1, 1]$ corresponding to indices $[2, 3, 1]$
- $[1, 1, 0]$ corresponding to indices $[3, 1, 2]$
- $[1, 0, 1]$ corresponding to indices $[3, 2, 1]$

EXPLANATION:

Since the array consists of zeroes and ones only the product of a subarray can only be 1 or 0. The product of a subarray is 1 if and only if it consists of all ones. Each such subarray consisting of all ones contributes 1 to the final the answer. Therefore the problem is reduced to finding number of subarrays that consists of all ones over all the $N!$ orderings of the array A.
Let C_1 represent the number of ones (count of ones) in the array A. Then for each length len from 1 to C_1, we can find number of subarrays of length len consisting of only ones over all $N!$ orderings of the array A by using combinatorics as:

- Select len indices of ones from C_1 indices of ones: ${}^{C_1}C_{len}$.
- Total arrangements of this subarray of length len are $Factorial_{len}$
- Starting positions in the array for this subarray are $N - len + 1$
- Total arrangements for rest of the numbers in the array are $Factorial_{N-len}$.

The product of these four values is the number of subarrays (Their contribution to the answer) of length len consisting of only ones over all $N!$ orderings of the array A. Add the answer for each length from $len = 1$ to C_1 to get the final answer.

The Binomial coefficients can be precalculated to improve the runtime of the algorithm. For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Find the number of permutations of $1, 2, ...n$ that have exactly k inversions, modulo 2

QUICK EXPLANATION:

There is a closed form expression for the answer which is given by

$f(n,k) = \binom{n-k-1}{k} + \sum_{j=1}^{\infty}(-1)^j\binom{n-k-1-u_j-j}{k-u_j-j} + \sum_{j=1}^{\infty}(-1)^j\binom{n-k-1-u_j}{k-u_j}$

$\binom{n}{r} \mod 2$ is 1 if $n \ \& \ k = k$ and 0 otherwise.

There are only $O(\sqrt{n})$ terms in the above expression that are positive in the above formula and each can be evaluated in $O(1)$.

EXPLANATION:

Finding recurrence relation

Let $f(n,k)$ be the number of permutations of $1, 2, ...n$ with exactly k inversions.

We can obtain a recursive formula for f, by considering what the first element of the permutation

Suppose we have a permutation of size n, that has k inversions, and the first element is x.

Then there will definitely be atleast $x-1$ inversions, because numbers $1, 2, ...x-1$ will occur after x.

Now the other $k-(x-1)$ inversion will be among the last $n-1$ elements.

Can we consider the last $n-1$ elements as a permutaion of size $n-1$?

As long as the numbers are different, their exact values doesn't matter. For eg. the set of inversions of $\{5, 3, 1\}$ is exactly the same as $\{3, 2, 1\}$

This is because we can replace every number by it's position in the sorted array and obtain an identical inversion set.

So we've reduced the problem from size n, k to a problem of size $n-1, k-(x-1)$.

Now we can formulate the recurrence by iterating through x.

$f(n,k) = \sum_{x=1}^{n} f(n-1, k-x+1)$

Base case $f(1,k)$ is 1 if $k=0$ and 0 otherwise.

This can be easily implemented with dynamic programming and complexity $O(nk)$, which is sufficient for subtasks

Generating function

Generating functions are useful to derive some important results in combinatorics. Such as the one here.

From the recurrence relation, we can obtain an explicit formula for $f(n,k)$ using generating functions.
This is a common technique in combinatorics.

For a sequence a, the generating function is a polynomial defined as $GF(x) = \sum_{k=0}^{\infty} a_k x^k$

Let's consider the sequence $f(n,0), f(n,1), ...$, and the corresponding generating function F_n

$F_n = \sum_{k=0}^{\infty} f(n,k)x^k$

Let's revisit the recurrence relation

$f(n,k) = \sum_{t=0}^{n-1} f(n-1, k-t)$ (replacing x with $t-1$ here to avoid confusion with the variable x)

$\implies f(n,k)x^k = \sum_{t=0}^{n-1} f(n-1,k-t)x^k$

$\implies f(n,k)x^k = \sum_{t=0}^{n-1} (f(n-1,k-t)x^{k-t})x^t$

Let's do summation on both sides, from $k=0$ to ∞

$\implies \sum_{k=0}^{\infty} f(n,k)x^k = \sum_{k=0}^{\infty} \sum_{t=0}^{n-1} (f(n-1,k-t)x^{k-t})x^t$

Rearranging the summation of the right hand side,

$\implies \sum_{k=0}^{\infty} f(n,k)x^k = \sum_{t=0}^{n-1} (\sum_{k=0}^{\infty} (f(n-1,k-t)x^{k-t}))x^t$

The term on lhs is clearly F_n. Let's have a closer look at the term inside the second $\sum$.

We have $\sum_{k=0}^{\infty} (f(n-1,k-t)x^{k-t}$.

Replacing $k' = k-t$, we have, $\sum_{k'=-t}^{\infty} (f(n-1,k')x^{k'}$.

SInce t is always non-negative, the terms having $k' = -t$ to $k' = -1$ are going to be zero ($f(n,-ve)$ is always 0, so we can drop those terms are write $\sum_{k'=0}^{\infty} (f(n-1,k')x^{k'}$ instead, which is exactly F_{n-1}!.

Therefore, $F_n = F_{n-1} \sum_{t=0}^{n-1} x^t$

This is a simple recurrence that we can expand (using base case $F_1 = 1$).

$F_n = (1)(1+x)(1+x+x^2)...(1+x+x^2+...x^{n-1})$

Using geometric progression formula, we get,

$F_n = \{(1-x)(1-x^2)(1-x^3)...(1-x^n)\}(1-x)^{-n}$
Deriving the expression

Let's try to find the coefficient of x^k in the generating function,

$F_n = \{(1-x)(1-x^2)(1-x^3)...(1-x^n)\}(1-x)^{-n}$

The second term, is well known to be $\sum_{t=0}^{\infty} \binom{n+t-1}{t}(-1)^t x^t$

All we want is the coefficient of x^k in F_n.

We can make use of Euler's pentagonal number theorem to simplify this.

$(1-x)(1-x^2)...(upto\ \infty) = 1 + \sum_{s=1}^{\infty} (-1)^s (x^{\frac{3s(s+1)}{2}} + x^{\frac{3s(s-1)}{2}})$

Although there are only n terms in the first (and not ∞ like in the formula above), appending additional terms in the expansion doesn't affect coefficient of x^k (since $k \leq n$).

$f(n,k)$ is equivalent to coefficient of x^k in the expansion

$\{(1-x)(1-x^2)(1-x^3)...(1-x^n)...upto\ \infty\}(1-x)^{-n}$, which is the same as

$\{1+\sum_{s=0}^{\infty}(-1)^s(x^{\frac{3s(s+1)}{2}}+x^{\frac{3s(s-1)}{2}})\}\sum_{t=0}^{\infty}\binom{n+t-1}{t}x^t$

There are only a few exponents in the first term which are less than n, for each exponent u in the first term, we can multiply $k-u$ in the second term.

This gives the following formula for $f(n,k)$

$f(n,k)=\binom{n-k-1}{k}+\sum_{j=1}^{\infty}(-1)^j\binom{n-k-1-u_j-j}{k-u_i-j}+\sum_{j=1}^{\infty}(-1)^j\binom{n-k-1-u_j}{k-u_i}$
ncr modulo 2

We can easily find the exponent of 2 in $n!, r!, (n-r)!$.
Using this we can find exponent of 2 in $\binom{n}{r}$ in $\mathrm{O}(\log n)$

We can obtain a faster algorithm using Lucas Theorem 3 , which is

$\binom{n}{r} \mod p = \binom{n_0}{r_0}\binom{n_1}{r_1}...\ upto\ \infty \mod p$ where p is prime.

where $n_0, n_1, ...$ are digits of n in base p and $r_0, r_1, ...$ are digits of r in base p.

Substituting for $p=2$, each digit is either 0 or 1.

Also, $\binom{0}{0}=\binom{1}{0}=\binom{1}{1}=1$ and $\binom{0}{1}=0$

If rhs in lucas theorem is 1 then every term must be 1, which means whenever k has a set bit, the corresponding bit in n is set.

This can be checked for all digits with bitwise operators directly, both the following are equivalent $n\ \&\ k == k$ and $n \mid k == n$ and are equal to $\binom{n}{r} \mod 2$

TIME COMPLEXITY:

TIME: $\mathrm{O}(\sqrt{n})$
SPACE: $\mathrm{O}(1)$

PROBLEM:

You are selecting two numbers x and y such that 1 <= x <= N and 1 <= y <= M. How many (x, y) pairs satisfy the condition that x + y is odd.

QUICK EXPLANATION

- Use the fact that Number of even numbers from 1 to N are floor(N / 2) and number of odd numbers from 1 to N are ceiling(N / 2).
- Answer will be (floor(N / 2) * ceiling(N / 2)) / (N * M).
- For representing the fraction in lowest terms, divide both the numerator and denominator by gcd of both.

EXPLANATION

Notice the following simple facts.

- Sum of an even and an odd number is odd.
  ie
  Even + Odd = Odd.
  Odd + Even = Odd.
- Number of even numbers from 1 to N are floor(N / 2).
  Number of odd numbers from 1 to N are ceiling(N / 2).

Fact number 1 is very easy to prove.

Let us prove fact 2, first part.
Claim Number of even numbers from 1 to N are floor(N / 2).
Proof:
We will prove this fact by using induction over N.

Base Case:
For N = 1, floor(1 / 2) = 0. Hence LHS = 0
As number of even numbers are 0 too from 1 to 1. Hence RHS = 0
So LHS = RHS

Induction Hypothesis:
Let us assume that formula is true up to N such that N is **even**, then we need to prove the formula for N + 1.
Notice that N + 1 will be odd. Hence number of even numbers won't increase and will remain equal to floor(N / 2).
As we know if N is even, then floor(N / 2) = floor((N + 1)/ 2). Hence we are done for this case.

Now Let us assume that formula is true up to N such that N is **odd**, then we need to prove the formula for N + 1.
Notice that N + 1 will be even. Hence number of even numbers increase by 1, hence they will be equal to floor(N / 2) + 1.
As we know if N is odd, then floor(N / 2) + 1 = floor((N + 1)/ 2). Hence we are done for this case too.

Note that instead of proving it formally, you can just see the above observation by observing the pattern for small numbers.

Proof of second part of fact 2 (Number of odd numbers from 1 to N are ceiling(N / 2).) will also go exactly along the same lines.

So finally our answer will be (floor(N / 2) * ceiling(N / 2)) / (N * M). We need to print this fraction in its irreducible form, so we have to divide both
the numerator and denominator by their gcd.

- Computing floor(N / 2) can be done by using simply integer division N / 2.
- Computing ceiling(N / 2) can be done by using simply integer divide (N + 1) / 2.

Complexity
O(log(N * M)), in the computation of gcd.

PREREQUISITES:

Derangement, next_permutation or random shuffle

PROBLEM:

You are given a permutation P of length N. A permutation of length N is an array where every element from 1 to N occurs exactly once.

You must perform some operations on the array to make it sorted in increasing order. In one operation, you must:

- Select two indices L and R $(1 \le L < R \le N)$
- Completely *dearrange* the subarray $P_L, P_{L+1}, \ldots P_R$

A dearrangement of an array A is any permutation B of A for which $B_i \neq A_i$ for all i.

For example, consider the array $A = [2, 1, 3, 4]$. Some examples of dearrangements of A are $[1, 2, 4, 3]$, $[3, 2, 4, 1]$ and $[4, 3, 2, 1]$. $[3, 5, 2, 1]$ is not a valid dearrangement since it is not a permutation of A. $[1, 2, 3, 4]$ is not a valid dearrangement either since $B_3 = A_3$ and $B_4 = A_4$.

Find the minimum number of operations required to sort the array in increasing order. It is guaranteed that the given permutation can be sorted in atmost 1000 operations.

HINT

The answer is always less than or equal to 2

EXPLANATION:

If the permutation P is already sorted the answer is 0. Let us look at the case when we can sort the permutation by just one move. If there exists a subarray $P[L, R]$ such that for each index i in the prefix $P[1, L-1]$, $P_i = i$ and for each index j in the suffix $P[R+1, N]$, $P_j = j$ and for all index k in the subarray $P[L, R]$, $P_k! = k$. Then we can derange the subarray $P[L, R]$ to achieve the sorted premutation. Otherwise we can always do just 2 operations to obtain the sorted permutation.

By Brute Force we can check that for all permutations of size 4 or greater there exists a derangement such that for no index i, $P_i = i$ in the resulting derangement. Now, apply the operation on the whole permutation P. To obtain the derangement of P such that for no index i $P_i = i$ in the derangement we break the array into size 4 consecutive subarrays starting from the index 1. Merge the last subarray to the second last subarray if it has a size less than 4. Now iterate over all permutations possible of the elements of these 4 size subarrays and check whether the permutation is a derangement as well as for no index i in the array we get $P_i = i$ by the derangement. This can be easily implemented using the next_permutaion and single for loop in C++.
The only case that remains is when $N = 3$.The only edge case in which an answer exists and it is not possible to get the answer using the approach defined above is $P = [3, 2, 1]$. This case can be solved beforehand and taken into account.

Answer for P={3,2,1}

2
$1\ 2$
$2\ 3\ 1$

1 3
1 2 3

TIME COMPLEXITY:

$O(N)$ or $O(N^2)$ depending on implementation for each test case.

PREREQUISITES:

Fast Fourier transform, Dynamic Programming, Divide and Conquer, Combinatorics, Modular multiplication

PROBLEM:

Alice initially had X stones. She divides the X stones into N piles where the ith pile contains A_i stones.

Now, she randomly selects a subset of piles S such that every subset has an **equal** probability of getting selected. After selecting the subset, for every $1 \le i \le N$,

- If $i \in S$, Alice colours all the stones in the ith pile RED.
- If $i \notin S$, Alice colours all the stones in the ith pile BLUE.

Let the total number of RED coloured stones be R and the number of BLUE coloured stones be B.
Alice now wonders what the expected value of $|R - B|$ is. Can you help Alice? Output the answer modulo 998244353.

Note that, $|x|$ denotes the absolute value of x. For example, $|-4| = 4$ and $|7| = 7$.

EXPLANATION:

The problem can be rephrased in simple words as:
Given an array A of length N, we randomly choose a subsequence P and add all the remaining elements to a second subsequence Q. Find expected value of $|sum_P - sum_Q|$ (where sum_X for a subsequence X represents the sum of all the elements of X and $|i|$ represents the absolute value of a number i). Find the answer modulo 998244353.

Let S be the sum of all elements of A (i.e. $S = \sum_{i=1}^{N} A_i$). Since each subsequence has equal probability of getting selected therefore for each subsequence we calculate $|S - 2 \cdot sum_P|$ and in the end divide the sum by 2^N to get the final answer.
The solution in steps:

Method 1: Brute force approach - Iterate over all subsequences P of array A and for each subsequence P formed calculate sum_P and $sum_Q = S - sum_P$ .The time complexity of this approach is $O(2^N)$ which is too slow for such large N.

Method 2: Dynamic Programing

The dp state is defined as follows: dp_i represents the number of ways to get sum equal to i by picking some subset of elements of array A. This dp can be calculated in $O(N * S)$ time using standard knapsack approach.
Example of calculating this dp

Then the final answer can be calculated as follows: $ans += dp_i * |S - 2 \cdot i|$ for all $0 \le i \le S$
But since N and S are both large, this is not fast enough to pass the test cases for the given constraints.
Sub-task for Method 2: FFT
We can use FFT to speed up the computation of the dp states. FFT is an algorithm that allows us to multiply two polynomials of length N in $O(NlogN)$ time,
Let the polynomial $P(x) = (1 + x^{A_1}) * (1 + x^{A_2}) * (1 + x^{A_3}) * ... * (1 + x^{A_n})$. We can compute P by multiplying the polynomials in a divide and conquer fashion using FFT. The coefficient of x^i in P represents the number of ways to

get sum equal to i.
Time complexity of this solution is $O(Slog(N)log(S))$ which is fast enough to pass the test cases.

For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(Xlog(X)log(N))$ for each test case.

PROBLEM

You are given three positive integers N, M and K.

An array of integers $A_1, A_2, \ldots, A_N$ is *good* if the following statements hold:

- $0 \le A_i \le M$ for each $(1 \le i \le N)$
- $(A_i \& K) \le (A_{i+1} \& K)$ for each $(1 \le i \le N-1)$
- $(A_i \mid K) \le (A_{i+1} \mid K)$ for each $(1 \le i \le N-1)$

Here, $\&$ denotes the bitwise AND operation and $\mid$ denotes the bitwise OR operation.

Count the number of good arrays $A_1, A_2, \ldots, A_N$. As the result can be very large, you should print it modulo $998\,244\,353$.

EXPLANATION

First increament M by 1 so we have to consider only $0 \le A_i < M$.

Small Observation

Let us consider only first 21 bits.
Let P contains the set bits in K, while Q contains the unset bits of K.
Then the array should be non-decreasing with respect to the bits present in P as well as Q.

General Approach while solving this problem

As we want $A_i < M$ so we will choose some prefix of M and mismatch the prefix only at the last index so that the later bits can be dealt independently.
For example if the binary representation of M is 110110, then we will consider the prefixes $1, 11, 1101, 11011$ and now changing the last bit we get $0, 10, 1100, 11010$.
Note that we have left the prefixes $110, 110110$ as we cannot make them smaller by changing the last bit.
Now after fixing the prefix S of length x we will ensure that the prefix of length x of every A_i should be $\le S$ as well as prefix of A_N (which is the largest element of the array) should be exactly S (so that we don't do any overcounting later).
Now after fixing some prefix S and maintaining the above inequalities we have established that all the A_i will be $< M$ so we can independently fill the remaining bits of the suffix.

Now after fixing some prefix S we need to count the number of good Arrays A such that

- prefix of $A_i \le S$
- prefix of $A_N = S$
- A is non-decreasing with respect to the bits of sets P and Q defined earlier.

After finding these counts for every prefix we can directly add them to obtain the answer. There will be no overcounting as we have fixed the prefix of A_N everytime.

Now let us consider $M = 1011100111$ in binary. Here Red color denotes the bits involved in set P, while Green color denotes the bits involved in set Q.

Let us consider the prefix of length 5 which is 10111. Now changing the last bit of that prefix we get $S = 10110$. Let us compress the bits of both colors in the prefix now. So for Red bits in the prefix we need to ensure that Red bits of

every A_i should be ≤ 11 (i.e. 3 in decimal). For Green bits in the prefix we need to ensure that Green bits of every A_i should be ≤ 010 (i.e. 2 in decimal). As there are no inequality restrictions with respect to M in the suffix so we just want that the Red bits value in suffix of A should be $\leq 2^3 - 1$ (Since there are 3 red bits in the suffix) and Green bits value in suffix of A should be $\leq 2^2 - 1$ (Since there are 2 green bits in the suffix).

Let's say that Red bits value and green bits value of the prefix of A_N be x and y respectively (In the taken example $x = 3$ and $y = 2$.
Let's say that Red bits value and green bits value of the suffix of A_N be p and q respectively (In the taken example $0 \leq p \leq 2^2 - 1$ and $0 \leq q \leq 2^3 - 1$).
We will first update x to $2^3 \cdot x$ (Since there are 3 red bits in the suffix) and update y to $2^2 \cdot y$ (Since there are 2 green bits in the suffix).
Now total Red value $= x + p$ and total green value $= y + q$.
We want Red value as well as Green value of A to be non-decreasing.
Let us fix the fix the Red Value of A_N to $x + p$ and Green Value of A_N to $y + q$, then the number of non-decreasing arrays satisfying the above conditions is:

$$\left(\binom{n+x+p}{n} - \binom{n+x+p-1}{n} \right) \cdot \left(\binom{n+y+q}{n} - \binom{n+y+q-1}{n} \right)$$

This formula can be derived using stars and bars. In the above formula the first term denotes the contribution of Red bits while second term denotes the contribution of Green bits. Also the term $\binom{n+x+p}{n}$ denotes the number of non-decreasing arrays of length n such that all the elements are $\leq x + p$ and ≥ 0. So subtracting $\binom{n+x+p-1}{n}$ we will get the count of non-decreasing arrays whose last term is $x + p$.

So for each prefix we will add the above formula to obtain the answer. See editorialists code for the above implementation.

Some common Mistakes

Keep your $MaxN = 4 \cdot 10^6$ for precompution of factorials.

TIME COMPLEXITY

$O(log^2 m)$ for every test case

PROBLEM:

You're given an array of integers A and a permutation P, each of size N.

The following operation is performed on the array K times:

- For each index i $(1 \le i \le N)$, add A_{P_i} to A_i **simultaneously**.

Find the values of A modulo 998244353, after performing the above operation K times.

QUICK EXPLANATION:

- Cycles of the permutation are independent.
- For each cycle, observe that the sequence of elements transform to their respective cyclically-shifted dot products with the sequence of binomial coefficients of order K. When the length of this sequence exceeds the size of the cycle, it wraps around and adds. Let's call this wrapped sequence a $pattern$.
- For a cycle of size m, we evaluate its $pattern$ by iterating over the elements of the binomial sequence in $O(K)$ time. We precompute $pattern$ for all relevant sizes.
- For a cycle of size m, the dot products of all its cyclic shifts with its $pattern$ can be computed modulo 998244353 using FFT (number theoretic transform) in $O(m \log m)$ time.

EXPLANATION:

Since each cycle of the permutation is independent, WLOG, let's solve for $P_i = (i\%N) + 1$.

It's easier to note the pattern from an example:

let $A = \{a, b, c, d\}$.

- iteration 0: $A = \{a, b, c, d\}$
- iteration 1: $A = \{a + b, b + c, c + d, d + a\}$
- iteration 2: $A = \{a + 2b + c, b + 2c + d, c + 2d + a...\}$
- iteration 3: $A = \{a + 3b + 3c + d, b + 3c + 3d + a...\}$
- iteration 4: $A = \{2a + 4b + 6c + 4d, ...\}$

(Note that binomial coefficients of order K are appearing and they start to wrap around the sequence. $(a + 4b + 6c + 4d + a)$)

The above pattern can be proven using induction and the following identity of binomial coefficients: $\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$

Formal Proof

We prove it inductively.

Let $V(i,j)$ be the value of A_i after j applications of the operation by the permutation P given as $P_i = (i\%N) + 1$. N is the size of A and P.
Proposition: $V(i,j) = \sum_{k=0}^{j} \binom{j}{k} A_{i+k} \forall i \in [1 \dots N]$ for all non-negative integers j.

Let $F(j)$ be the statement $V(i,j) = \sum_{k=0}^{j} \binom{j}{k} A_{i+k} \forall i \in [1 \dots N]$

Assume that for a particular j, $F(j)$ is true.
$\implies V(i,j) = \sum_{k=0}^{j} \binom{j}{k} A_{i+k} \forall i \in [1 \dots N]$

By definition, $V(i,j+1) = V(i,j) + V(i+1,j)$.
$\implies V(i,j+1) = \sum_{k=0}^{j} \binom{j}{k} A_{i+k} + \sum_{k=0}^{j} \binom{j}{k} A_{i+1+k}$
define $\binom{j}{-1} = \binom{j}{j+1} = 0$,
$\implies V(i,j+1) = \sum_{k=0}^{j+1} \binom{j}{k} A_{i+k} + \sum_{k=0}^{j+1} \binom{j}{k-1} A_{i+k}$
$\implies V(i,j+1) = \sum_{k=0}^{j+1} (\binom{j}{k} + \binom{j}{k-1}) \cdot A_{i+k}$
$\implies V(i,j+1) = \sum_{k=0}^{j+1} \binom{j+1}{k} A_{i+k}$

Thus, $F(j) \implies F(j+1)$. Since $F(0)$ is trivially true, $(V(i,0) = A_i)$, $F(j)$ is true for all non-negative j.

So for given K and size of the sequence L, we generate the pattern C as

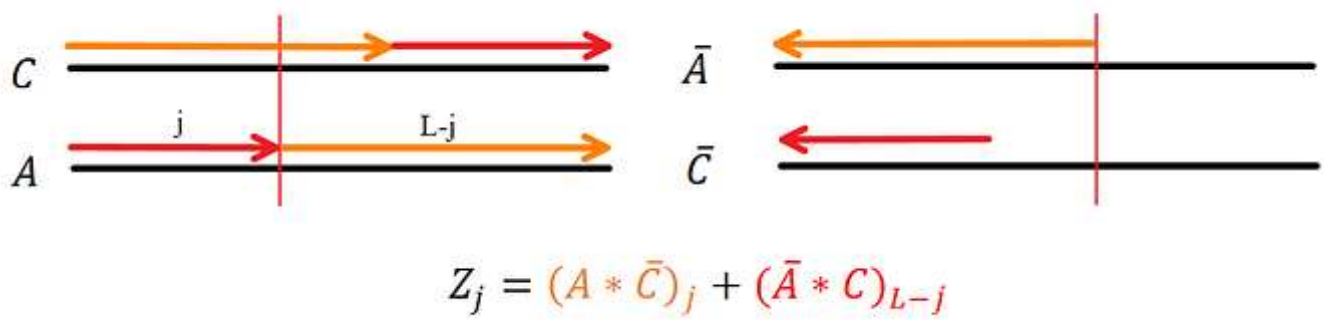
```
for i in 0..K:
        C[i % L] += ncr(K, i)
```

Since the size of the permutation is N, there are $O(\sqrt{N})$ different cycle sizes and this step takes $O(K\sqrt{N} + N)$ time.

Now we need to find the dot product of C and A over each cyclic shift.

$Z_j = \sum_{i=0}^{L-1} A_{(i+j)\%L} \cdot C_i$

which can be computed in $O(L \log L)$ time using FFT as $Z = (A * \bar{C})\%x^L + \overline{(\bar{A} * C)\%x^L}$, where $\bar{P}$ is the reversed polynomial of P and $*$ denotes convolution.



TIME COMPLEXITY:

$O(K\sqrt{N} + N \log N)$