PREREQUISITES:

Binomial coefficients

PROBLEM:

Alice wants to test Bob's love for the number 7.

Alice gives Bob a sequence of integers $A = [A_1, A_2, \ldots, A_N]$, and Q queries. In each query, she provides an integer K and asks bob to find the number of *lovely* subsequences of size K that can be formed from A.

A sequence B of length K is called *lovely* if, for every subsequence S of B, the following condition holds:

- If $|S|$ is divisible by 7, then the sum of S should also be divisible by 7.
- If $|S|$ is not divisible by 7, then the sum of S should also not be divisible by 7.

where $|S|$ denotes the length of S.

Help Bob answer the queries. Since the answers can be very large, print them modulo $10^9 + 7$.

Note: Two subsequences are considered distinct if the indices chosen to form them are distinct, even if their elements end up being the same. For example, $A = [1, 1, 2, 2]$ has 4 subsequences of length 1 (two are $[1]$ and two are $[2]$) and 6 subsequences of length 2.

EXPLANATION:

As we are concerned with only the remainder of the sum of numbers, for all index i from $i = 1$ to N, replace each element of the array A by A_i%7 and keep the count of all elements in the new array. Let Cnt_i represent the count of i in the modified array A. No index i with $A_i = 0$ can be a part of a *lovely* subsequence as the subsequence formed by taking only this element does follow the second condition.
Now, for $K > 7$ all *lovely* subsequences consist of exaclty 1 distinct number from 1 to 6.

Proof

Let us suppose there exists a *lovely* subsequence of A with length > 7 which contains at least 2 distinct numbers. Select a subsequence, from this *lovely* subsequence, of size 7 such that it contains at least 2 distinct numbers. The sum of this subsequence has to be divisible by 7 as this is a subsequence of size 7 of a *lovely* subsequence (first condition). Now, pick any element from the remaining element of the *lovely* subsequence and swap this with any element in the subsequence, we took, of size 7 which is not equal to it (which must exist as there are two distinct values in the subsequence) and this sum cannot be divisible by 7 now.
$\therefore$ for $K > 7$ any *lovely* subsequence contains no more than 1 distinct number.

For $K > 7$ iterate on values from i=1 to 6 and add $\binom{Cnt_i}{K}$ to the answer.

For $K <= 7$ we can pre-calculate the answer for each value of K from 1 to 7. We have 6 values for each element of the subsequence and this generates a total of $6^1 + 6^2 .....6^7$ different combinations (Since only count of each value is important sort all the combinations later). Iterate on each combination and check whether it is a *lovely* subsequence. The actual total number of lovely subsequences comes out to be very less (102). Now, for each *lovely* subsequence find the number of ways in which it can be constructed from A.
Let C_i represent the count of i in a *lovely* subsequence of size ≤ 7, then the total number of possible ways to

construct this *lovely* subsequence from A is given by $\prod_{i=1}^{6} \binom{Cnt_i}{C_i}$

Add this to the answer for a particular size K (equal to the length of this *lovely* subsequence).
Pre-calculations can de done for binomial coefficients to avoid increasing the runtime of the solution.

For details of implementation, please refer to the solutions attached.

TIME COMPLEXITY:

$O(N + Q)$ or for each test case.

PROBLEM:

There is a stone of mass 2^N, initially lying at the origin. For the next N seconds, each stone of mass M at position X splits into one stone of mass $\frac{M}{2}$ going to position $X-1$ and another of mass $\frac{M}{2}$ going to position $X+1$.

Given L and R, find for each position i such that $L \le i \le R$ how many stones lie at position i at the end of the process.

EXPLANATION:

Hint

Think of the stone of mass 2^N as being 2^N separate stones of mass 1. Encode the movement of each stone as a binary string of length N, distinct stones receiving distinct strings. Now relate this encoding to the final position.

Full solution

Instead of stones halving in mass, it becomes a bit easier to think of the problem as involving 2^N stones, each of mass 1. The given operation then essentially says that at each position, half the stones present there move left and the other half move right.

Now note that the movement of any given stone can be represented by a binary string of length N — the i-th character of the string is `1` if this stone moved right on the i-th move, and `0` if it moved left.

Under this setup, it's easy to see that no two stones will receive the same binary string, i.e, no two stones have the exact same movement pattern.

Proof

This can be proved by induction on N. If $N=1$ the result is trivial, since there are 2 stones, one moving left and the other moving right on the only move made.

For $N>1$, after the first move there are 2^{N-1} stones on -1 and 2^{N-1} on $+1$. Clearly, any pair consisting of a stone at -1 and a stone at $+1$ has different binary strings, since the very first character differs.

Now, consider only those stones at -1. They correspond to the process with $N-1$ and hence any two within this pile will also differ eventually, by the inductive hypothesis. Similarly, any two stones at $+1$ will also differ eventually.

With 2^N stones and 2^N binary strings, it's easy to see that there is a bijective mapping between stones and binary strings.

Now, we use this information to actually solve the problem at hand.
Consider a stone such that its binary string has x ones and y zeros.
It made x steps right and y steps left, which means it ended up at position $x-y$.

Let's turn that around. Suppose we fix a position k. How many stones ended up at position k?

Well, we know that if a stone made x steps right and y steps left and then ended up at k, we must have $x-y=k$. On the other hand, clearly we also have $x+y=N$, the total number of moves made.

Note that these equations give us a unique solution for the pair (x,y) — namely, $x=\frac{k+N}{2}$ and $y=\frac{N-k}{2}$. Of course, x and y do need to be integers, so if k and N have different parities then no stone can end up at position k.

Now, say we fix position k and calculate x and y. How many stones have binary strings corresponding to this x and y?
As established earlier, each binary string corresponds to a single stone and vice versa, which means that all we need to do is to compute the number of binary strings of length N consisting of x ones and y zeros!

A simple combinatorial argument shows that this number is just $\binom{x+y}{x} = \binom{N}{x}$ — choose which x of the N positions are ones, and the others are automatically fixed to contain zeros.

Note that the binomial coefficient needs to be calculated modulo $10^9 + 7$, so if you don't know how to do that a tutorial has been linked above.

TIME COMPLEXITY:

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

PREREQUISITES:

Combinatorics, Dynamic Programming, Sliding Window

PROBLEM:

Given a permutation array of length N. Determine the number of ways to assign a direction (left or right) to each number such that it is possible to sort the permutation by applying the following operation any number of times - Select any element of the permutation and move it to the left/right end of the array, based on the assigned direction.

EXPLANATION:

Let $f(i)$ represent the number of *good* assignments, where the integers (**NOT indices**) $1, 2, \ldots, i$ are assigned direction `left` and integer $i+1$ is assigned direction `right`.
Then, the answer is equal to $f(0) + f(1) + \cdots + f(N)$.

Let us determine $f(i)$ for a fixed i.
Since we assign the first i integers direction `left`, we can sort them by moving them to the left in increasing order.

Then, we only concern ourselves with sorting the remaining integers.
Let $pos[x]$ represent the index of integer x in the permutation array.

Hint 1

If $pos[i+2] < pos[i+1]$, integer $(i+2)$ must be assigned direction `right` (Why?)

Generalising, if for any x, $pos[x+1] < pos[x]$ and integer x is assigned direction `right`, all integers j $(j > x)$ must be assigned direction `right` (the reasoning is similar as above and easily deducible).

Hint 2

If $pos[i+1] < pos[i+2]$, integer $(i+2)$ can be assigned either `left` or `right`.

Why?

On extrapolating, it is easy to see that, if $pos[i+1] < pos[i+2] < \cdots < pos[i+k]$, then all integers $(i+1), (i+2), \ldots, (i+k)$ will never have to be operated on, and hence can each have their direction set to either `left` or `right`.

Solution

Let S_x represent the greatest k such that $pos[x] < pos[x+1] < \cdots < pos[x+k]$. This array can be computed efficiently using sliding window.

Then, from hint 2, it is clear that each of the integers $(i+2), (i+3), \ldots, (i+S_{i+1})$ can be assigned either `left` or `right` (independent of other assignments). Let p represent the number of such integers.

From hint 1, it is then clear that all other integers $(i+S_{i+1}+1), \ldots, N$ can only be assigned direction `right`.

Then, the total number of *good* assignments equals

$$f(i) = 2^p$$

which can be determined in $O(1)$ with suitable precomputations!

TIME COMPLEXITY:

Calculating array S can be done using sliding window in $O(N)$. Computing $f(i)$ can be done in $O(1)$ by precomputing the powers of 2. Since we calculate $f(i)$ for each valid i, the total time complexity is

$$O(N + N) \approx O(N)$$

per test case.

PREREQUISITES:

Combinatorics, Inclusion Exclusion, FFT/NTT

PROBLEM:

Given two positive integers N and M. Let S be the set of all binary grids of dimensions $N * M$, where the coordinates of the 1's in the grid are strictly increasing.

Let $f(T)$ represent the number of ways to place rooks in the grid T such that, the rooks are non-attacking and no rook is placed in a cell with value 1.

Determine $\sum f(t) \mod 998244353$, where $t \in S$.

EXPLANATION:

It is immediately obvious that the coordinates of the 1's in each of the generated grids are strictly increasing. Hence, each grid has atmost one 1-valued cell in each row/column.

Let $P = \min(N, M)$. It is also obvious that the maximum number of rooks over all valid arrangements is equal to P.

Hint 1

Start by finding the number of valid ways to place rooks in a $N * M$ grid consisting of 0's only.

Solution

Since there are no *unavailable* cells in the grid, we are only concerned with finding the number of different placements such that no two rooks are attacking. This is a classical problem, whose solution can be represented by the formula:

$$\sum_{i=0}^{P} \binom{N}{i} \cdot \binom{M}{i} \cdot i!$$

In brief, we count the number of ways to place exactly i rooks (summed over all valid i), by selecting i distinct X and Y coordinates and calculating the number of different sets of coordinates that can be created by pairing these values.

Hint 2

Now solve for when there is exactly one 1-valued cell in the grid. A little intuition tells us that **the answer is unaffected by the coordinates of the 1-valued cell(s)**.

Solution

Subtracting the *number of ways to place non-attacking rooks where one rook is placed on the 1-valued cell* **from the** *total number of ways to place non-attacking rooks*, gives us the desired answer.

How do we calculate the former mentioned value?

The number of ways to place non-attacking rooks on a $N * M$ grid, with the position of one rook fixed, is equivalent to the number of ways to place non-attacking rooks on a $(N-1) \cdot (M-1)$ sized grid (*Why?*)

More formally, let W_i represent the number of ways to place non-attacking rooks on a grid of size $(N-i)\cdot(M-i)$. Then, the answer to the above question is equal to

$$W_0 - W_1$$

Can you extend this to solve for when there are i $(1 \le i \le P)$ 1-valued cells?

Solution

Standard application of inclusion-exclusion.

The answer is equal to (derivation is mundane and left to the reader as an exercise):

$$\binom{i}{0}W_0 - \binom{i}{1}W_1 + \binom{i}{2}W_2 - \cdots + \binom{i}{i}W_i$$

Hint 3

So far, we've derived a method to determine the number of ways we can place rooks when there are exactly $1 \le i \le P$ cells with value 1. Call this value Q_i.

Then, if K_i represents the number of valid grids with *exactly* i 1-valued cells, the answer to the problem is equal to:

$$\sum_{i=0}^{P} K_i \cdot Q_i$$

How will you calculate array K?

Solution

By our initial observation (mentioned at the start), the following equation is trivially derivable:

$$K_i = \binom{N}{i} \cdot \binom{M}{i}$$

However, we are not done yet. The time complexity of the described solution is $O(P^2)$, which will TLE in the final subtask. Clearly, we need to optimise!

Analysis (optional, but recommended)

We summarise our progress, along with the time complexity of our approach.

K_i represents the number of valid grids with exactly i, 1-valued cells.

$$K_i = \binom{N}{i} \cdot \binom{M}{i}$$

which can be calculated in $O(P)$ with suitable pre computations.

W_i represents the number of ways to place non-attacking rooks on a grid of size $(N-i)\cdot(M-i)$.

$$W_i = \sum_{k=0}^{P-i} \binom{N-i}{k} \cdot \binom{M-i}{k} \cdot k!$$

and can be computed naively in $O(P^2)$ time.

Q_i represents the number of ways we can place rooks when there are exactly i cells with value 1.

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

that similarly has an $O(P^2)$ time complexity.

The final answer is then -

$$\sum K_i \cdot Q_i$$

The total time complexity is therefore:

$$O(P + P^2 + P^2 + P) \approx O(P^2)$$

Hint 4

A huge part of our algorithm is devoted to multiplying and summing the values of different arrays (implicitly). This just screams FFT/NTT.

Can you rewrite the above equations to utilise the power of FFT?

Solution

Let's start with efficiently calculating array W.

$$\begin{aligned} W_i &= \sum_{k=0}^{P-i} \binom{N-i}{k} \cdot \binom{M-i}{k} \cdot k! \\ &= (N-i)! \cdot (M-i)! \cdot \sum_{k=0}^{P-i} \frac{1}{k! \cdot (N-i-k)! \cdot (M-i-k)!} \end{aligned}$$

Now, the summation of the products (last term of the equation) can be calculated at once over all i using FFT.

What are the coefficients of the polynomials to multiply

Similarly, we can rewrite Q_i and apply FFT on it. Deriving the suitable formula is left to the reader as an exercise (you may have to simplify $\sum K_i \cdot Q_i$ and then apply FFT, which is what my solution does). Refer my (well written) code for help.

TIME COMPLEXITY:

Computing array K takes $O(P)$. Computing array W takes $O(P \log P + P)$. Calculating the final solution takes $O(P \log P + P)$. The final time complexity is therefore:

$$O(P \log P)$$

PREREQUISITES:

Bitwise OR, Combinatorics

PROBLEM:

Let's define $Score$ of an array B of length M as $\sum_{i=1}^{M} \sum_{j=i}^{M} f(i,j)$, where $f(i,j) = B_i \mid B_{i+1} \mid \ldots \mid B_j$ (Here $\mid$ denotes the bitwise OR operation)

You are given an array A, consisting of N **distinct** elements.

Find the sum of $Score$ over all $N!$ possible permutations of A.

Since the answer can be large, output it modulo 998244353.

QUICK EXPLANATION:

- Contribution of ith bit in the *Score* is independent of the contribution of jth bit, for $i \neq j$.
- Sum of *Scores* over all permutations can be written as - sum of contribution of ith bit (for $1 \leq i \leq 30$) over all permutations.
- Find total number of subarray (over all permutations) such that the ith bit is not set in the OR of these subarrays. Using this, calculate number of subarrays num_i such that the ith bit is set in the OR of these subarrays.
- Contribution of the ith bit = $2^{i-1} \cdot num_i$, where $num_i = (N+1)! \cdot (\frac{N}{2} - \frac{K}{N-K+2})$ and K represents the number of numbers in which the ith is not set.

EXPLANATION:

Score of an array is defined as the sum of the OR of all the possible subarrays. Now, an important observation regarding OR (which holds for other bitwise operators too) is that the result of the ith bit is independent of the result of the jth bit. This observation suggests us that if we can focus on a fixed bit, and can calculate the contribution of that bit in the *Score*, then we can solve our problem.

Let us focus on the ith bit. Let num_i denotes the number of subarrays such that in the OR of these subarrays, the ith bit is set. This is equivalent to saying that at least one element in each of these subarrays have the ith bit set. Now, it is easier to calculate the number of subarrays in which none of the elements have the ith bit set. Hence, we can write num_i as follows:

num_i = Total number of subarrays - number of subarrays in which none of the elements have the ith bit set

Total number of subarrays

If we consider a single permutation, then the number of subarrays in that permutation is $\frac{(N)(N+1)}{2}$.
Total number of permutations = $N!$, where ! represents factorial.
Hence, total number of subarrays = $N! \cdot \frac{(N)(N+1)}{2}$

number of subarrays in which none of the elements have the i-th bit set

Let K represents the number of numbers in which the ith bit is not set.
We will calculate the required number of subarrays by calculating the subarrays of length len $(1 \leq len \leq K)$ for

each len individually. Note that all the subarrays of length $len \geq K+1$ will have at least one element with the ith bit set.

Now, the number of subarrays of length len satisfying the required property
$= \binom{K}{len} \cdot (len)! \cdot (N-len)! \cdot (N-len+1)$

First term denotes the number of ways to choose len elements out of the K elements which have the ith bit as not set. The second term denotes the number of ways to arrange these len elements and third term denotes the number of ways to arrange the remaining elements. The fourth term denotes the number of positions where this subarray of length len can be placed in the entire array.

Summing this expression for all the values of len

After simplifying, the above expression can be written as

$(K)! \cdot (N-K+1)! \cdot \binom{N-len+1}{N-K+1}$

Summing over all values of len, we get

$\sum_{len=1}^{K} (K)! \cdot (N-K+1)! \cdot \binom{N-len+1}{N-K+1}$

$= (K)! \cdot (N-K+1)! \cdot \sum_{len=1}^{K} \binom{N-len+1}{N-K+1})$

The summation term can be represented as the sum of coefficients of x^{N-K+1} in $(1+x)^{N-len+1}$.
Coefficient of x^{N-K+1} : $(1+x)^{N-K+1} + (1+x)^{N-K+2} + \cdots + (1+x)^{N}$.

We can first simplify the Geometric Progression at the Right Hand Side, and say that the above value is equivalent to coefficient of x^{N-K+2} in $(1+x)^{N+1}$.

Substituting this value back in the original expression, we get the final value as $\frac{(N+1)! \cdot K}{N-K+2}$

Hence, $num_i = (N+1)! \cdot (\frac{N}{2} - \frac{K}{N-K+2})$

Contribution of the ith bit = $2^{i-1} \cdot num_i$. So finally, we need to sum up this contribution over all values of i to get the final answer.

TIME COMPLEXITY:

We can pre-compute all the factorials beforehand : $O(N)$
Then for each test case, calculate K for each i - $O(N \cdot \log A_i)$, and use these values to get answer for each i.
Overall Time Complexity: $O(N \cdot \log A_i)$ for each test case.

PREREQUISITES:

Combinatorics, Graph Theory

PROBLEM:

You are given an array A of length N such that $1 \le A_i \le N$ and an integer M. Count the number of arrays B of length N, such that

- $B_i \equiv B_{A_i}$
- $1 \le B_i \le M$

Since the answer can be large, print it modulo $10^9 + 7$.

QUICK EXPLANATION:

- Build a graph with N edges as (i, A_i). The graph will have some connected components.
- Each connected component has exactly one cycle.
- Calculate the answer for the cycle. Each non-cycle node can be filled in $(M - 1)$ ways. The total answer for a component is the product of the answer for cycle as well as that of non-cycle nodes.
- Final answer is the product of the answers of all connected components.

EXPLANATION:

Observation

We are given that $B_i \equiv B_{A_i}$. To connect the ith position with the $A_i{}^{th}$ position, we can build a graph with N edges. Each edge is generated using the vertices i and A_i.

Considering $1 \le A_i \le N$, the graph would consist of various connected components. Each of the connected components will have exactly one cycle. In other words, the graph would be a tree with one extra edge which forms a cycle.

Proof

Any connected component of size K will have exactly K edges. For more than one cycle to exist, there has to be atleast $K + 1$ edges. Also, for no cycles, there can't be more than $K - 1$ edges. Thus, there is exactly one cycle present.

For each connected component, let us look at this cycle part separately.

Subproblem

Problem: Find number of arrays B of length $N(\ge 2)$ such that:

- $B_1 \equiv B_N$
- $B_i \equiv B_{i+1}$ $(1 \le i < N)$
- $1 \le B_i \le M$

Solution: We need to find the number of arrays such that no two consecutive elements have the same value (including first and last element) and each element has value in range $[1, M]$. We can use dynamic programming to calculate this.

Let $dp[i][0]$ denote the number of ways to fill first i elements such that $B_j \neq B_{j+1}$ $(1 \leq j < i)$ and $B_1 \neq B_i$. Similarly, let $dp[i][1]$ denote the number of ways to fill first i elements such that such that $B_j \neq B_{j+1}$ $(1 \leq j < i)$ and $B_1 = B_i$. The answer to our problem is the value $dp[N][0]$.

- **Base Case:** $dp[1][0] = 0$, there is no way in which we can fill the first element such that it is not equal to the first element. Similarly, $dp[1][1] = M$.
- **Calculating** $dp[i][1]$**:** Since there is only one way of filling the ith element, $dp[i][1]$ is nothing but $dp[i-1][0]$.
- **Calculating** $dp[i][0]$**:** If the $(i-1)^{th}$ element is equal to the first element, the ith element can be filled in $(M-1)$ ways (all numbers except the value of the first element). Else, there are $(M-2)$ ways to fill that (all numbers except the value of the first and the $(i-1)^{th}$ element). Thus, $dp[i][0] = (dp[i-1][1] \cdot (M-1) + dp[i][0] \cdot (M-2))\%mod$.

Conclusion

Consider a connected component with X nodes. Let this component contain a cycle of length Y. For the cycle part, the answer is $dp[Y][0]$. For all other nodes, which are not part of the cycle, there are $(M-1)$ ways to fill each one of them. Thus, the answer for this component is $(dp[Y][0] \cdot (Y-X)^{M-1})\%mod$.

The final answer would be the product of the answers of all such components.

TIME COMPLEXITY:

The time complexity is $O(N)$ per test case.

PROBLEM:

You are given two numbers N and M. A pair of arrays (A, B) is called K-beautiful if and only if:

- $|A| = |B| = N$ (i.e. both their lengths are equal to N)
- For all i such that $1 \le i \le N$, $0 \le A_i \le M$ and $0 \le B_i \le M$.
- $mex(A) \le mex(B)$
- A is lexicographically smaller than B
- The first index i where $A_i < B_i$ is K

For each K ($1 \le K \le N$), find the number of K-beautiful pairs and print those values modulo MOD.

EXPLANATION:

We first loop over K, then loop over the exact value of $mex(A)$ (which I will denote as MEX), and finally over the exact number of distinct values that are less than MEX and appear in the common prefix $A_{1..K-1}$ (or $B_{1..K-1}$).

Denote S_1 as the set of distinct values that are less than MEX and appear in the common prefix, S_2 to be the set of distinct values that are less than MEX and **have yet** to appear in the common prefix, S_3 to be the set of values larger than MEX, and S_4 to be the set of values larger or equal to MEX (this means $S_4 = S_3 \cup \{MEX\}$, and $S_1 \cup S_2 = [1..MEX - 1]$). The sizes of these sets can be calculated easily. We then have the following observations:

- Since we want the mex value of A to be exactly MEX, the suffix $A_{K..N}$ must contains all values within S_2. Similarly, since we want the mex value of B to be at least MEX, the suffix $B_{K..N}$ must contains all values within S_2.
- Every element in A that is not in S_1 or S_2 must be in S_3 (since we want $mex(A)$ to be exactly MEX).
- Every element in B that is not in S_1 or S_2 must be in S_4 (since we want $mex(B)$ to be at least MEX), except for the prefix $B_{1..K-1}$ where it must comes from S_3 (since this is the common prefix).
- Any value in S_1 or S_2 is less than any value in S_3 or S_4.

This leads us to loop over which sets A_K and B_K can be in. It turns out that there are 7 cases:

- $A_K \in S_1, B_K \in S_1$.
- $A_K \in S_1, B_K \in S_2$.
- $A_K \in S_1, B_K \in S_4$.
- $A_K \in S_2, B_K \in S_1$.
- $A_K \in S_2, B_K \in S_2$.
- $A_K \in S_2, B_K \in S_4$.
- $A_K \in S_3, B_K \in S_3$.

I will guide you through the first case, but other cases can be reasoned similarly. The number of ways to have $A_K \in S_1, B_K \in S_1$ is:

$$\binom{MEX}{|S_1|} \cdot \frac{|S_1| \cdot (|S_1| - 1)}{2} \cdot f(K-1, |S_1|, |S_3|) \cdot f(N-K, |S_2|, |S_1| + |S_3|) \cdot f(N-K, |S_2|, |S_1| + |S_4|)$$

where $f(i, j, k)$ is the number of ways we can fill an array with i elements, in which we have j values that are “fixed”, and for each element not having a fixed value there are k ways to fill in its value. To explain the formula above:

- $\binom{MEX}{|S_1|}$ comes from choosing the set S_1 from $[0..MEX - 1]$.

- $\frac{|S_1|\cdot(|S_1|-1)}{2}$ comes from choosing $A_K < B_K$.
- $f(K-1, |S_1|, |S_3|)$ comes from filling the common prefix such that it contains all values in S_1, and any elements not in S_1 comes from S_3.
- $f(N-K, |S_2|, |S_1|+|S_3|)$ comes from filling $A_{K+1..N}$ such that it contains all values in S_2, and any elements not in S_2 comes from $S_1 \cup S_3$.
- $f(N-K, |S_2|, |S_1|+|S_4|)$ comes from filling $B_{K+1..N}$ such that it contains all values in S_2, and any elements not in S_2 comes from $S_1 \cup S_4$.

We have yet to cover how to compute $f(i,j,k)$, but it turns out that this a pretty easy task (reminiscence of counting the number of arrays with n elements and k distinct values):

$$f(i,j,k) = f(i-1,j-1,k) + (j+k)\cdot f(i-1,j,k).$$

That is because there are 3 states that can reach $f(i,j,k)$:

- Add a value that has already been added from the fixed values, this can be done in j ways and comes from state $f(i-1,j,k)$.
- Add a new fixed value, which comes from state $f(i-1,j-1,k)$.
- Add a non-fixed value, this can be done in j ways and comes from state $f(i-1,j,k)$.

The above reccurence won't really calculate what we wanted, since the first appearance of the fixed values is unordered. Therefore, after the recursion, we need to multiply $f(i,j,k)$ by $j!$.

TIME COMPLEXITY:

Time complexity is $O(M\cdot N\cdot \min(M,N))$

PROBLEM:

You are given an array A of length N, which initially is a permutation of integers from 1 to N.
Find the number of ways to perform N operations wherein each operation we choose any index i and remove A_i from A (which decrements the size of A by 1) ensuring that the array doesn't contain any consecutive monotone triples. Print answer modulo $10^9 + 7$.

EXPLANATION:

Hint 1

At any point, we can choose to remove one amongst at max 4 points, 2 from the start, and 2 from the end.

Hint 2

Try coming up with an expression involving the last 3 elements left in terms of the removed $N - 3$ elements.

Hint 3

Suppose the indexes of the last 3 remaining elements are represented as `L`, `M` and `R`, the number of ways to get:
`L` as the left of the 3 remaining elements is independent of indexes belonging to the set $\{M+1, M+2, ..., N\} - \{R\}$ and
`R` as the right of the 3 remaining elements is independent of indexes belonging to the set $\{1, 2, ..., M-1\} - \{L\}$.

Solution

Suppose after the removal of x elements our array is represented as A_x and the size of A_x = sz_x. We can choose to remove one amongst at max 4 points to reach A_{x+1}:

1. First (,i.e, A_{x_1}) (Easy to see)
2. Second (,i.e, A_{x_2})
3. Last (,i.e, $A_{x_{sz_x}}$) (Easy to see)
4. Second last (,i.e, $A_{x_{sz_x-1}}$)

Proof

Let the present state of the array be represented as B.
Also, note that B must not contain any monotone triples. Consider the first 5 elements of B which must be related as:

1. $B_1 > B_2 < B_3 > B_4 < B_5$ or,
2. $B_1 < B_2 > B_3 < B_4 > B_5$.

Why other elements can't be removed

Suppose the first relation holds (proof for the second will be similar).
After removing B_3 one of the following relations will hold:

1. $B_1 > B_2 > B_4 < B_5$ or,
2. $B_1 > B_2 < B_4 < B_5$.
   both of which have a 3 monotonic triple. Same will hold for other elements as well.
   So we have a contradiction, so we can't remove anything except for the mentioned 4 elements.

When second and second last elements can be removed

Let's talk about the second element, the same will follow for the second last element as well.
So this holds when for condition:

1. $B_1 < B_3$
2. $B_1 > B_3$

Now B_3 becomes our second element which can be removed when for condition:

1. $B_1 > B_4$
2. $B_1 < B_4$

...

Terminating Condition

This removal is possible until some index x where for condition:

1. $B_1 < B_{x+2}$
2. $B_1 > B_{x+2}$

when x is even, and

1. $B_1 > B_{x+2}$
2. $B_1 < B_{x+2}$

when x is odd.
,i.e, we can't remove index $x+1$ without removing index 1.

Suppose the indexes of the last 3 remaining elements after removing the remaining $N-3$ elements are represented as `L`, `M` and `R`, the number of ways to get:
`L` as the left of the 3 remaining elements is independent of indexes belonging to the set $\{M+1, M+2, ..., N\} - \{R\}$ and
`R` as the right of the 3 remaining elements is independent of indexes belonging to the set $\{1, 2, ..., M-1\} - \{L\}$.
which means contribution from the removal of a prefix is independent of the contribution from the removal of the suffix.

So we can fix our `M` and find the number of ways to get (`L`, `M`, `R`) as the last 3 elements as follows:
Iterate from index(i) 2 to $N-1$ and consider it as our `M`.
Find the number of ways in which we can get an index $j < i$ as `L` and
let the sum of the number of ways over all valid j:

1. Where $A_j < A_i$ be `mulL[i][0]`.
2. Where $A_j > A_i$ be `mulL[i][1]`.

Find the number of ways in which we can get an index $k > i$ as `R` and
let the sum of the number of ways over all valid k:

1. Where $A_k < A_i$ be `mulR[i][0]`.
2. Where $A_k > A_i$ be `mulR[i][1]`.

Finally the triplet (j, i, k) is valid if either both A_j and A_k are smaller than A_i or vice-versa.

Our answer = $(\sum_{i=2}^{N-1}$ ((`mulL[i][0]` $\cdot$ `mulR[i][0]` + `mulL[i][1]` $\cdot$ `mulR[i][1]`) $\cdot(\binom{N-3}{i-2})$)) $\cdot 3!$.

where $\binom{N-3}{i-2}$ represents the number of ways to choose $i-2$ positions where we remove an index $i <$ `M` out of the total $N-3$ removals and $3!$ represents the arrangement of last 3 remaining elements since they can be removed in any order.

We explain the computation part only for prefixes, computation for suffixes can be done similarly by reversing the array.

Calculating mulL array

Suppose we represent removing the first element as `op1` and removing the second element as `op2`.
Define `ways[i]` as the number of ways to remove the prefix of length i.
Define `range[i]` as maximum index $j(> i)$ such that if index i is the first element, elements in the range $[i+1, i+2, \ldots, j]$, can be the second element if we only choose `op2` and not `op1`.
So:
`ways[0]` = 1
`ways[i]` = `ways[i - 1]` + $\sum_{j=1}^{i-1}$ `ways[j - 1]` (where $j < i$ and `range[j]` $\geq i$). (for $i \geq 1$)

- First term - After removing $i-1$ length prefix, perform `op1` operation.
- Second term - After removing $j-1$ length prefix, perform $i-j$ `op2` operations and then perform `op1` operation.

Calculating range array

Define:
`forward[i][0][0]` as the smallest even index $j > i$ such that $A_j < A_i$.
`forward[i][1][0]` as the smallest odd index $j > i$ such that $A_j < A_i$.
`forward[i][0][1]` as the smallest even index $j > i$ such that $A_j > A_i$.
`forward[i][1][1]` as the smallest odd index $j > i$ such that $A_j > A_i$.

if no such index exists, value = `INF`.

`range[i]` = min(min(`forward[i][i & 1][1]`, `forward[i][1 - (i & 1)][0]`) - $2, N$)

The expression comes from our previous discussion that how long we can keep removing the second element.
We can calculate the `forward` array using the segment tree in $\mathrm{O}(N \log_2 N)$ and the `range` array in $\mathrm{O}(N)$ after that.

Calculating ways array

`ways` array can be calculated in $\mathrm{O}(N)$ using prefix sums and range update query.

Finally,

$$\texttt{mulL[i][0]} = \sum_{\substack{j=1 \\ A_j < A_i}}^{i-1} \texttt{ways[j - 1]}$$

$$\texttt{mulL[i][1]} = \sum_{\substack{j=1 \\ A_j > A_i}}^{i-1} \texttt{ways[j - 1]}$$

which can be calculated in $\mathrm{O}(N \log_2 N)$ using segment tree after calculating the `ways` array and in $\mathrm{O}(N)$ using prefix sums and range update query.

ALTERNATE IMPLEMENTATION:

Check the tester's code for reference.

COMPLEXITY ANALYSIS:

So final complexity is $\mathrm{O}(N \log_2 N)$.

PREREQUISITES:

Combinatorics, Number-theoretic transform (NTT)

PROBLEM:

You are given an array A of N integers, and an integer X. For each K in the range $[1, N]$, determine the number of ways to partition the array into **exactly** K non-empty subarrays such that the maxima of each of the subarrays are **at least** X.

The number of ways can be large, so output it modulo 998244353.

QUICK EXPLANATION:

To partition an array into K non-empty subarrays, we have to choose one starting point, $L \in [1, N]$, corresponding to each subarray. Therefore we require k starting points, $L_1, L_2, L_3, ..., L_k$.

- We can reduce the original array into a binary array by keeping all the elements greater than or equal to X as 1 and remaining as 0 s.
- If the positions of 1 in A are $\{p_1, p_2, p_3, ..., p_m\}$, then there are $(p_{i+1} - p_i)$ ways to select one starting point and 1 way to select no starting point in the subarray $[p_i + 1, p_{i+1}]$ $\forall i \in [1, m]$.
- We can model a subarray into the polynomial $(p_{i+1} - p_i) \cdot x^1 + 1 \cdot x^0$, where the powers of each term denote the number of starting points taken and the coefficients represent the number of ways to select that many starting points.
- We can then find the total number of ways to select starting points among all such subarrays through the polynomial $\prod_{i=1}^{m-1}((p_{i+1} - p_i) \cdot x^1 + 1 \cdot x^0)$. The resulting polynomial will be of the form $C_0 \cdot x^0 + C_1 \cdot x^1 + ... + C_{m-1} \cdot x^{m-1} + 0 \cdot x^m + ... + 0 \cdot x^{n-1}$.
- The total number of ways for K-partition will be the coefficient of x^{k-1} in the resulting polynomial. Note that the initial starting point L_1 is fixed to be the index 1 of A.

We can compute the resulting polynomial by multiplication of individual polynomials, with the help of NTT. The time complexity for this approach will then remain $O(N \log N)$.

EXPLANATION:

Easier Version

Let's first solve an easier version of this problem, where we only have to find the **number of ways** in which we can partition the array A into subarrays such that the maximum of every partitioned subarray is **at least** X. Informally, this easier problem asks us to sum the answer over all Ks belonging to the original problem.

In order to solve this, we notice that to create a new partition of a non-empty subarray we only have to choose one starting point, L. We then look at this problem as finding these starting points, $L_1, L_2, L_3, ..., L_k$, over all the $[1, N]$ elements of the array. Finding k starting points will give us K subarrays. Also, note that the initial starting point, L_1, will correspond to the starting position, i.e., index 1.

Without loss of generality, we can reduce the original array into a binary array by keeping all the elements greater than or equal to X as 1 and remaining as 0 s.

Now, let the positions of 1 in A be $\{p_1, p_2, p_3, ..., p_m\}$. The subarray $[p_i + 1, p_{i+1}]$ $\forall i \in [1, m]$ will look like: $00..01$. This subarray of length $(p_{i+1} - p_i)$ will have $(p_{i+1} - p_i - 1)$ number of 0 s followed by a single 1. Our task requires us to calculate the number of ways to select starting points among all such subarrays.

A subarray can only have **at max one** starting point. Since if there were more, then at least one partition will be of all 0 s, and that partition will remain invalid since all the elements in the original array were less than X.
If the subarray has zero starting points, then this whole subarray will belong to the partition which was started before it, at some point L_j in some previous subarray. Elsewhen we choose one starting point at any index j, call it L_j, where $j \in [p_i + 1, p_{i+1}]$, then this subarray will be divided into two parts, where the left part will belong to the previous partition starting at L_{j-1} and the right part will belong to the newly generated partition starting at L_j.

Thus, there are $(p_{i+1} - p_i)$ ways to select one starting point and 1 way to select no starting point in a subarray. In total, one subarray can contribute to $(p_{i+1} - p_i + 1)$ number of ways. Now, the total number of ways to select starting points among all such subarrays will be $\prod_{i=1}^{m-1}(p_{i+1} - p_i + 1)$. Note that the initial starting point L_1 is fixed to be the index 1 of A, and it belongs to the subarray $[1, p_1]$ which is not considered in this product.

Original Version

Considering the original version of the problem, we are to report the individual number of ways for K-partition, where $K \in [1, N]$. We can do this using the polynomial method in combinatorics.

For a subarray $[p_i + 1, p_{i+1}]$, we have gathered that there are $(p_{i+1} - p_i)$ ways to select one starting point and 1 way to choose no starting point. We can model this into the polynomial $(p_{i+1} - p_i) \cdot x^1 + 1 \cdot x^0$, where the powers of each term denote the number of starting points taken and the coefficients represent the number of ways to select that many starting points.

We can then find the total number of ways to select starting points among all such subarrays with $\prod_{i=1}^{m-1}((p_{i+1} - p_i) \cdot x^1 + 1 \cdot x^0)$. The resulting polynomial will be of the form $C_0 \cdot x^0 + C_1 \cdot x^1 + ... + C_{m-1} \cdot x^{m-1} + 0 \cdot x^m + ... + 0 \cdot x^{n-1}$. Here, the answer for K of our problem will be the coefficient of x^{k-1}. Note again that the initial starting point L_1 is fixed to be the index 1 of A, and it belongs to the subarray $[1, p_1]$ which is not considered in this product polynomial.

In order to find the resulting polynomial, we can regard the distinct values of $\{p_{i+1} - p_i; \forall i \in [1, m]\}$ as $\{v_1, v_2, ..., v_t\}$. Let the frequencies of $\{v_1, v_2, ..., v_t\}$ in $\{p_{i+1} - p_i; \forall i \in [1, m]\}$ be $\{f_1, f_2, ..., f_t\}$ respectively. Since $\sum_{i=1}^{t}(v_i \cdot f_i) < N$, t has an upper bound of $O(\sqrt{N})$.

Now, the polynomial $\prod_{i=1}^{m-1}((p_{i+1} - p_i) \cdot x^1 + 1 \cdot x^0)$ can be represented as $\prod_{i=1}^{t}(v_i \cdot x^1 + 1 \cdot x^0)^{f_i}$. In other terms, $\prod_{i=1}^{t}(\sum_{j=0}^{f_i}\binom{f_i}{j} \cdot v_i^j \cdot x^j)$.
We can compute for this polynomial as a multiplication over $O(\sqrt{N})$ small polynomials of the form $\sum_{j=0}^{f_i}\binom{f_i}{j} \cdot v_i^j \cdot x^j$, that are of length f_i. The multiplication of these polynomials can be performed with Number-theoretic transform (NTT). To multiply all the individual polynomials we can sequentially pick two polynomials, multiply in $O(d \cdot \log d)$, where d is the sum of degrees of the two polynomials, and keep the merged result for further multiplications. This way we can reach the final polynomial: $C_0 \cdot x^0 + C_1 \cdot x^1 + ... + C_{m-1} \cdot x^{m-1} + 0 \cdot x^m + ... + 0 \cdot x^{n-1}$, where total number of ways for K-partition will be represented by the coefficient of x^{k-1}.

We also have to take care of two corner cases:

1. **All** elements of A are smaller than X:
   In this case, $\{p_1, p_2, p_3, ..., p_m\}$ will be an empty set, $\emptyset$, i.e., $m = 0$.
   The answer for $K \in [1, N]$ will remain 0.

2. **Only one** element of A is greater than or equal to X:
   In this case, $m = 1$ and $\{p_1, p_2, p_3, ..., p_m\}$ will contain a single element, i.e., the position of the only 1 element. For this, the only starting point will remain $L_1 = 1$.
   Thus, the answer for $K = 1$ will be 1, whereas for $K \in [2, N]$ the answer remains 0.

TIME COMPLEXITY:

The calculation of the sets $\{p_1, p_2, p_3, ..., p_m\}$, $\{v_1, v_2, ..., v_t\}$, and $\{f_1, f_2, ..., f_t\}$ shall all cost $O(N)$ time.

Now, we want to calculate $\prod_{i=1}^{t}(v_i \cdot x^1 + 1 \cdot x^0)^{f_i}$. We know that $\sum_{i=1}^{t}(v_i \cdot f_i) \leq N$ and $v_i \in [1, N)$ To compute the polynomial, let's introduce j to replace v_i, and assume f_j corresponds to $f_i : j = v_i$. The equation $\sum_{j=1}^{N}(j \cdot f_j) \leq N$ still holds. Then we can reframe the polynomial as $\prod_{j=1}^{N}(j \cdot x^1 + 1 \cdot x^0)^{f_j}$.

The computation of individual polynomials of the form $\sum_{i=0}^{f_j}\binom{f_j}{i} \cdot j^i \cdot x^i$ shall cost $O(f_i)$ time using binomial theorem, i.e. $O(N)$ time in total.

We can multiply the polynomial in reverse order:

```
Res = Poly(1);
for (int j = N; j > 0; j--)
    Res = Res * genBinom(j, f_j);
```

i-th iteration works in $O((f_i + f_{i+1} + \ldots + f_N)\log N)$ time, so complete operation will take $O(\log N \sum_{j=1}^{N}(j \cdot f_j)) = O(N \log N)$ time. This same approach can be found here [3] .

The time complexity for the given approach is thus $O(N \log N)$.

PREREQUISITES:

Expected Values, Combinatorics

PROBLEM:

Chef, who is a chef by profession as well, has created N pies (enumerated 1 through N) and has placed them in a row. Now he wants to taste all the pies, but by following a rather peculiar order of tasting.

At each step, he chooses (uniformly randomly and independently) either the current leftmost pie or the rightmost pie, and eats that pie.

For example, if $N = 7$, then one valid order of tasting the pies is $(1, 2, 7, 3, 4, 6, 5)$. However, $(1, 2, 7, 6, 4, 5, 3)$ is not a valid order.

Chef can't wait to taste all the pies, so for each i $(1 \leq i \leq N)$, he wants to know the expected value of the number of steps he'll need to eat the ith pie.

It can be shown that each expected value can be expressed as a fraction $\frac{P}{Q}$. You should compute $P \cdot Q^{-1}$ modulo $10^9 + 7$, where Q^{-1} denotes the modular multiplicative inverse of Q under $10^9 + 7$

EXPLANATION:

Let's denote the operation of choosing leftmost pie by L, and choosing rightmost pie by R.

Let us consider the Pth pie from the left, and try to calculate the Expected value for this pie.
There are two ways to pick this pie - either using an L operation, or using an R operation.

Let us focus on the case when the Pth pie was chosen using an L operation.

A Natural, but incorrect(?) approach

Assume that this pie was chosen after $P + K$ operations. This means that number of L operations will be P, and the number of R operations will be K. Also, the last operation should be L.

Number of ways to choose pie after $P + K$ operations such that the last operation is L is same as Number of ways to arrange $\#(P - 1)$ Ls and $\#K$ Rs. Also, Because choosing L and R are equiprobable, the probability of each such arrangement is $(\frac{1}{2})^{P+K}$.

Therefore, the Probability of choosing P^{th} pie after $P + K$ operations such that the last operation was L = $\binom{P-1+K}{K} \cdot (\frac{1}{2})^{P+K}$.

And the Expected value can be calculated as:

$$E_P = \sum_{K=0}^{N-P} \binom{P-1+K}{K} \cdot (\frac{1}{2})^{P+K} \cdot (P + K)$$

Notice the constraints on K. The maximum number of R operations can be $N - P$, as we want the Pth pie to be chosen by an L operation.

The issue with the above summation is, it is difficult to calculate its value efficiently.
Do let us know in the comments if you have been able to calculate it efficiently!

A Magical, working approach

In this approach, we will continue choosing the pies even after we have chosen the Pth pie. This complete simulation can be represented as a string of length N having L and R as characters.

Let say that in the complete process, the R operations were done a total of K times.
Also, let say that the R operations were done a total of i times before choosing the Pth pie using L operation.

Because the Pth pie is chosen using L operation, the maximum value of K can be $N - P$.

Total number of strings such that the Pth pie is chosen using L operation will be

$$S_P = \sum_{k=0}^{N-P} \sum_{i=0}^{k} \binom{P-1+i}{i} \cdot \binom{N-(P+i)}{k-i} \cdot (P+i)$$

On Simplifying

$$S_P = P \cdot \sum_{k=0}^{N-P} \binom{N+1}{k}$$

Note that the sum S_P has currently accounted only for the case when the Pth element is chosen using an L operation.
To account for the case when P^{th} element is chosen using an R operation, we can observe that P^{th} element from the left is $(N - P + 1)^{th}$ element from the right, and therefore the contribution coming from this case will be same as the contribution from $(N - P + 1)^{th}$ element from the left (because L and R operations are symmetric).

Therefore, the final value of S_P will be:

$$S_P = (P \cdot \sum_{k=0}^{N-P} \binom{N+1}{k}) + ((N - P + 1) \cdot \sum_{k=0}^{P-1} \binom{N+1}{k})$$

Each term can be calculated efficiently using the running sum.

Finally, each string is equiprobable with probability $(\frac{1}{2})^N$. Therefore, the expected value for the Pth pie can be represented as:

$$E_P = (\frac{1}{2})^N \cdot S_P$$

TIME COMPLEXITY:

The above algorithm can be implemented in $O(N \cdot \log N)$ as well as $O(N)$ depending on the pre-computation and implementation details.

PREREQUISITES:

HASHING, DYNAMIC PROGRAMMING, COMBINATORICS

PROBLEM:

Given an Array of Integers , Find the number of ways you can split the array into continuous subarrays such that distinct elements in one subarray does not occur in another.

EXPLANATION:

In order to find the total number of ways to split the array with the given constraint, let us first find the splitting of the array A into K continuous subsets $S1$, $S2$, …, Sk for some arbitrary K, such that $S1$ + $S2$ + … + Sk = A and no subset Si (1 <= i <= K) can be split further and is the smallest possible. Note that the $'+'$ symbol denotes the concatenation symbol here.

To find this, let's define an map M, which stores the position of the last occurrence of all distinct elements in the array. This can be be achieved in linear time by using Hashing. Next step is to define a DP array P of length N such that $P[i]$ stores the position of the last occurrence of the element $A[i]$ i.e; $M[A[i]]$. Now the main purpose to use a DP array is to find the all the spitting points in the array.

Now the question is how to achieve this?

For simplicity, let us consider a single subarray $S[x:y]$ (1 <= x <= y <= N) such that the elements of this subarray does not belong to another subarray, which means if an element $e \in S[x:y]$, the all the occurrences of the element e in the array should lie in the range $[x, y]$. The converse is also true, if an element $t \notin S[x:y]$, the all the occurrences of the element e in the array does not occur in the range $[x, y]$.

But what to do with these analysis?

Lets take an example,
consider an array $A = [1, 2, 1, 3, 1, 2, 5, 4, 5, 6, 7, 7]$.

We know that the smallest splitting for this array would be $[1, 2, 1, 3, 1, 2]$, $[5, 4, 5]$, $[6]$, $[7, 7]$.

The map M of this array will be M = { $1:4, 2:5, 3:3, 5:8, 4:7, 6:9, 7:11$ } (with '0' based indexing).

Then the DP array $P = [4, 5, 4, 3, 4, 5, 8, 7, 8, 9, 11, 11]$.
It can be observed that the certain numbers fall in a particular range, these ranges are our required subsets. To find the range let us define an array T where $T[i]$ stores the maximum element among the first i elements in P.
Then our T array is,
$T = [4, 5, 5, 5, 5, 5, 8, 8, 8, 9, 11, 11]$
It can be noted here that if $T[i]$ equals i, then the index i is one of the splitting point.

By this we can find all the minimal subarrays of any given array.

Now that we have all the K subarrays, how to find the total number of possible splitting?

We have K subarrays with at this point. Now we can observe that even if we concatenate any two adjacent subarrays, the resulting splitting of the subarrays would still be not violating the constraint that any distinct element should be present in at most one subarray. Since we have K subarrays which means there are $K-1$ possible concatenations that you can perform. Every choice of either concatenating or not concatenating for each of the $K-$

1 possibilities would result in a possible split. Hence by using combinatorics, we can conclude that there would be 2^{K-1} total possibilities to split the array.

PREREQUISITES:

Combinatorics

PROBLEM:

We are given N children and their goodness level $A[i]$. Santa will distribute some coins to each of the children. If two children have the same goodness level, then they must receive the same number of coins. If one of the children has a greater goodness level, then that child should receive more coins than the child with a lesser goodness level. We have to output the number of ways in which Santa can distribute the coins. Also, we need print the distribution in which Santa spends the most coins.

EXPLANATION:

Since two children with the same goodness level must receive the same number of coins, the number of distinct integers in the distribution of Santa's coins must equal to the number of distinct numbers in the goodness level of N children.

Santa has a total of N options for the array B as $1 \leq B[i] \leq N$. Let R be the number of distinct integers in the array A. Then the total number of ways in which Santa can distribute the coins is $\binom{N}{R}$.

Of these, the one in which Santa spends the most number of coins is where he gives N coins to the child/children with the highest goodness level, $N-1$ coins to the child/children with the second-highest goodness level, and so on.

We can use a vector of pair where the first part stores the goodness level and the second part stores the index number. We can sort this vector and make the required B array.

Time Complexity : $O(NlogN)$

PREREQUISITES:

Math, Fermats's Little Theorem, Combinatorics

PROBLEM:

Given a sequence A of length N, the weight of this sequence is defined as $weight(A) = \sum_{i=1}^{N} i \cdot A_i$. Given an integer N, we pick a permutation A of $[1, 2 \ldots N]$ uniformly at random. What is the expected weight of this permutation? It can be shown that this can be represented as P/Q where P and Q are co prime integers and $Q \not\equiv 0 \bmod 998244353$. Print the value of $P \cdot Q^{-1}$ modulo 998244353.

EXPLANATION:

$weight(A)$ for a permutation A from 1 to N as mentioned in the problem is given by -

$$weight(A) = 1 \times A_1 + 2 \times A_2 + \cdots + n \times A_n$$

This in turn is calculated for all possible permutations A, let the evaluation of the value $weight(A)$ for the i_{th} permutation be w_i (where $0 < i \leq N!$). We are required to calculate the expected value of weight of such a randomly picked permutation, this will be given by -

$$E = (w_1 + w_2 + \cdots + w_{n!})/(n!)$$

If we fix the i_{th} element of A to a value j (where $0 < j \leq N$), there will be $(N-1)!$ permutations possible by changing positions of other elements. In simpler words, there will be $(N-1)!$ weights in E whose corresponding permutation A had the number j at the i_{th} position, meaning $i \times j$ is contributing its value $(N-1)!$ times to E.

For example consider $N = 3$, in this case if we fix $A_i = j$ for $j = 1$ and i from 1 to N we observe -

1. $i = 1, j = 1$
   $A_1 = 1$ -
   $A = [1, 2, 3]$, $weight(A) = 1 + 2 \times 2 + 3 \times 3$
   $A = [1, 3, 2]$, $weight(A) = 1 + 2 \times 3 + 3 \times 2$

2. $i = 2, j = 1$
   $A_2 = 1$ -
   $A = [2, 1, 3]$, $weight(A) = 2 + 2 \times 1 + 3 \times 3$
   $A = [3, 1, 2]$, $weight(A) = 3 + 2 \times 1 + 3 \times 2$

3. $i = 3, j = 1$
   $A_3 = 1$ -
   $A = [2, 3, 1]$, $weight(A) = 2 + 2 \times 3 + 3 \times 1$
   $A = [3, 2, 1]$, $weight(A) = 3 + 2 \times 2 + 3 \times 1$

We can observe that for each value of i from 1 to N, $i \times j = i \times 1$ is present in the weight of $(N-1)! = (3-1)! = 2$ permutations of A.

Now we've established that each number j will appear at a position i $(0 < i \leq N)$, $(N-1)!$ times. Each number j from 1 to N can contribute $i \times j$ to the expected value $(N-1)!$ times. We can write it as follows -

$$1 \times i \times (N-1)! + 2 \times i \times (N-1)! + 3 \times i \times (N-1)! + \ldots N \times i \times (N-1)!$$
$$= i \times (N-1)! \times (1+2+3+\cdots+N)$$
$$= i \times (N-1)! \times ((N)(N+1)/2)$$

where i can assume values between 1 to N for each value of j as well, we can write this as -

$$1 \times (N-1)! \times ((N)(N+1)/2) + 2 \times (N-1)! \times ((N)(N+1)/2) + 3 \times (N-1)! \times ((N)(N+1)/2) + \ldots N \times (N-1)! \times ((N)(N+1)/2)$$
$$= (1+2+3+\cdots+N) \times (N-1)! \times ((N)(N+1)/2)$$
$$= (N \times (N+1)/2) \times (N-1)! \times ((N)(N+1)/2)$$
$$= (N! \times N \times (N+1)^2)/4$$

We have above obtained the sum of weights of all $N!$ permutations of A, to calculate the expected value E now we need to divide this sum by the number of permutations possible -

$$(N! \times N \times (N+1)^2)/(4 \times N!)$$
$$= (N \times (N+1)^2)/4$$

Out of the components of the answer above, $N \times (N+1)/2$ will always yield an integral quotient which is why we won't be taking the modular multiplicative inverse of 4 that is currently in the denominator. We can divide it into 2 parts -

1. The part yielding integral quotient
2. The part that may or may not leave a remainder on division

as follows -

$$(N \times (N+1)/2)$$
$$\times$$
$$(N+1)/2$$

If we compare it to P/Q form where P and Q are co prime integers, we can claim that either

1. $P = N \times (N+1)^2/2$ and $Q = 2$ or
2. $P = N \times (N+1)^2/4$ and $Q = 1$

Now if $N+1$ is found to be divisible by 2, we can simply multiply P by $1^{-1} modulo (998244353)$. On the other hand if P is not divisible by 2, we will have to multiply it with $2^{-1} modulo (998244353)$. This leads us to our final answer -

$(N(N+1)^2/2) \times 2^{-1} modulo (998244353)$ if N is even, or
$(N(N+1)^2/4) \times 1^{-1} modulo (998244353)$ if N is odd

TIME COMPLEXITY:

$O(1)$ per test case, because computing modular multiplicative inverse i.e. $Q^{-1} modulo(998244353)$ takes $O(log(998244353))$ time which is a constant.

PROBLEM IN SHORT:

Given a number n, how many permutations of numbers-
1,2,…,n-1,n are there such for every k < N the set of ministers sitting in the first k chairs DOES

NOT consist only of the ministers 1, 2, . . . k.

Each digit(0-9) has a cost.

What is the maximum number you can write such that the total cost incurred is less than equal to B?

The cost for each digit will be given to you.

QUICK EXPLANATION:

Let ans[n] denote the answer for that n.
Then calculate all the values till n = 4 manually.
After that
ans[5] = 5!-(ans[1]*4!+ans[2]*3!+ans[3]*2!+ans[4]*1!)
ans[6] = 6!-(ans[1]*5!+ans[2]*4!+ans[3]*3!+ans[4]*4!+ans[5]*5!)
and so on …

EXPLANATION:

Let ans[n] denote the answer for a that n
Now we can manually calculate the values till n = 4
So after doing some pen and paper work we get,
ans[1] = 1
ans[2] = 1
ans[3] = 3
ans[4] = 13
Now let's compute ans[5] smartly without all the manual work,
So the total number of permutations not excluding the ones not allowed are 5!
Now lets keep subtracting the ones not allowed
So, the first number cannot be 1. So if the first number is 1 and the other numbers no matter whatever they might be we have to exclude them. So, total number of permutations with first number being 1 = 4! because we can choose the rest 4 numbers in any order.
Now the first two numbers cannot be a permutation of 1,2
which is basically 1,2 and 2,1 but we notice that we can't subtract the permutation starting with 1,2 as we have already subtracted the ones starting with 1. So we subtract the ones starting with 2,1 and the rest three numbers can be anything so that's 3!.
Now the first 3 numbers can't be a permutation of 1,2,3
So they can't be starting with

1,2,3
1,3,2
2,1,3
2,3,1
3,1,2
3,2,1
We have already excluded the ones starting from 1 or 2,1
which leaves us
2,3,1
3,1,2
3,2,1
Here we also notice one thing,
that this cnt = ans[3] = 3 ([2,3,1],[3,1,2],[3,2,1])
the previous cnt was ans[2] = 1 ([2,1])
and the one before that ans[1] = 1 ([1])
and how is it exactly coming this way?
So, notice here ans[2] = count of all the permutations of length 2 such that they don't contain 1 as the first letter (1 was the permutation for ans[1])

and ans[3] = count of all the permutations of length 3 such that they don't contain 1 or 2,1 at start (1 was the permutation for ans[1] and 2,1 was the one for ans[2])
also ans[4] = count of all the permutations of length 4 such that they don't contain 1 or 2,1 or either of ([2,3,1],[3,1,2],[3,2,1]) at start (1 was the permutation for ans[1] and 2,1 was the one for ans[2] and the last 3 were those of ans[3])
So, let's forget all that we have done while calculating ans[5] and let's use this easier method. (I was telling you the other way so as to come to this point)
Also, note that we are not double counting as while removing let's say those of ans[2] we aren't removing those of ans[1] as ans[2] doesn't include those.
So, we can define ans[5] as 5! - (ans[1]*4!+ans[2]3!+ans[3]2!+ans[4]1!)*
= *120-(124+16+32+13*1)
= 120-(24+6+6+13)
= 120-49
= 71
Bonus: Do the 2nd and 3rd part yourself.
Hope that helped.

PROBLEM:

Given N distinct points on a plane, out of which M points lie on a single straight line $(M \leq N)$. There is only one such line on the plane which can contain more than 2 points. Count the number of distinct straight lines formed after connecting each point with all the other points. Print it modulo $10^9 + 7$.

QUICK EXPLANATION:

Combinatorics can help us solve this problem. We can prove that we will have $(\binom{N}{2} - \binom{M}{2} + 1)$ unique lines on this plane after connecting each point with the other one.

EXPLANATION:

Let us first consider all combinations of pairings of the points on this plane, mathematics always has the answer. First of all, there can be $\binom{N}{2}$ ways of choosing two points (for connection) out of the N points on this plane without repetition. Now it is also given that there are M points on a single straight line, therefore, we don't have to join those points among each other explicitly, because in that case we will be counting more than we need. Also, in that case, we will also count non-unique straight lines.

Therefore, we should consider the ways of connecting two points among those M points and reduce those combinations from our result for all of N points. This way we can eliminate repetition of lines (we need to print minimum possible lines without repetition). Also, if we are counting unique lines, then there is only one way of counting, and there is only one result, so we, in a way, always guarantee to find the minimum number of lines drawn on this 2-D plane (after all connections made).

Number of ways of choosing two points (for connection) out of the M points are $\binom{M}{2}$. Now, we subtract it from our result for N points and we get $\binom{N}{2} - \binom{M}{2}$. But, hey, we are forgetting something really important here. We subtracted the result for M points and we just lost all connectivity among those M points. Therefore, we need to add 1 to this result, because there is exactly one way to connect all the M points together (a single straight line).

$\therefore$ we have proved that our answer is going to be $(\binom{N}{2} - \binom{M}{2} + 1)$ mathematically using combinatorics and some geometry about straight lines. Now comes the implementation part.

Sub-Task 1

For the constraints of sub-task 1, we can use a brute force approach by actually counting the ways one by one for each of the points (iterating over all other points). Time complexity will be $O(N^2)$. This way we get 20 points.

Sub-Task 2

Here, we can build a factorial prefix using modular arithmetic, and we can find factorial inverse under modulo of a prime number (Let $K = 10^9 + 7$) using Fermat's Little Theorem. Therefore time and space complexities would be $O(N + \log_2 K)$ and $O(N)$ respectively. This way, we are actually finding the values for the binomial coefficients modulo K. This way we will get 40 points (sub-task 1 and sub-task 2).

$\{O(N)$ time for building prefix, and $O(\log_2 K)$ time for finding modular inverse using Fermat's Little Theorem $\}$

Sub-Task 3

By just applying simple mathematics, we can easily figure out that $\binom{N}{2} = \frac{N \times (N-1)}{2}$. Therefore we brought down linear time to constant time. $\therefore$ time complexity now is $O(1)$, fair enough to pass the tests.

Note: Apply modular arithmetic very carefully because here we have a *subtraction* involved. So, take modulo only once and like this:

$$(\frac{N\times(N-1)}{2} - \frac{M\times(M-1)}{2} + 1) \bmod (10^9 + 7)$$

COMPLEXITY:

$O(1)$ time per test case, with $O(1)$ auxiliary space used.

Question:

The Salesman is playing Ddakji with Seong Gi-Hun.
There are N cardboards kept at distinct positions, the ith of which can be flipped A_i **times only**. Gi-Hun gets 100 won with each flip. He has to play the game Q times and can select a series of **consecutive** positions to flip the cardboard kept at those positions .
The Salesman being clever, has already decided the series of positions, which Gi-Hun would have to select in all the games.
Since Gi-Hun is in huge debts, he should now stealthily arrange the cardboards in a manner so as to receive the **maximum total sum of money**.

Before actually arranging the cardboards, you must help him find the total number of distinct arrangements possible to obtain the maximum sum. Since the value can be large, find its modulo $10^9 + 7$.
Note: Gi-Hun would only select the **indices** that The Salesman has already decided, even if their values are not the same as before.

Two arrangements are considered different if there exists at least one position where their values are not the same in both the arrangements.

Quick Explanation:

Since the indices which would be selected is given to us, we must place the largest element in the most frequently occuring index, second largest to the second most occuring index and so on…
Suppose multiple indices are selected equal number of times, placing the correspondingly large values in either place would give the same total.
Hence all we have to do is to check for the multiple positions with same frequency of getting selected, and permuting them, with the values we can place at those positions.
Note: Permuting equal elements wouldn't make a new arrangement, even if multiple indices have the same frequency.

Detailed Explanation:

Our first task is to find the frequency of the indices. Since we are given a series of consecutive indices, we can use [prefix sums](#).

short description

We can create a new array/vector named fre, with every element initialised with 0.
Given L,R for each query, we would increment the Lth index of fre with 1, and decrement $(R+1)^{th}$ (if it exists) index by 1.
The value of $fre[i]$ would be obtained by successively adding the vaues uptil that position.
The resultant value at the ith position would give the frequency of the ith index.

After having obtained the frequencies of the indices in the fre vector, we also have to keep track of the number of indices which have the same frequency.
We can define a vector $hash$ to keep a track of this.
Now how to find the number of arrangements ?
Suppose we have 4 indices with the same value- The number of ways of arranging 4 **distinct** elements in those positions would be $4!$. But what if 2 elements have the same value?
We can say that permuting those 2 elements would not cause any change in the number of distinct arrangements that we should obtain. Thus, the number of arrangements now would be $4!/2!$.

Therefore our final answer would be $\prod_{i=1}^{n}(fact[hash[i]])/(fact[non_distinct_in_hash[i]])\%(100000007)$
$fact$ denotes the precomputed factorial array.

As stated earlier, the largest frequencied should be placed at the most occuring indices - hence the count for distinct elements would be done on them.

Time Complexity

$O(NLog(N) + Q)$

- $NLog(N)$: for sorting (and if used map/set)
- Q : For taking the Q queries as inputs

PROBLEM:

Two people are fighting. We have to find the no. of fights possible where either of them dies in each round, and one of them can resurrect N times with a condition that he cannot die more than once consecutively, and the other one can resurrect M times with no condition.

EXPLANATION:

Note: Faizal cannot die twice consecutively, if Faizal dies, it is necessary for Faizal to win the next round to win the fight.

Let's first find the number of ways for Faizal to win, then we will find the no. of ways to lose, and add them.

For winning:

Let's iterate, over all the number of times Faizal can resurrect ($i = 0$ to n) ,

Ex: If $i = 4$, (let's denote Faizal's death by F and win by S, there will be at least one S just after F, as it is necessary for Faizal to win the next round so we group F and S together i times and find the number of ways to distribute S at the remaining places.)

$_FS_FS_FS_FS_$ (for $i = 4$, we have 5 places, where we have to place S, so that the total sum of S is equal to $m + 1$, as Sultan can resurrect m times, he must die $m + 1$ times for Faizal to win), so the number of ways to win is equal to number of ways to place distribute $(m + 1 - i)$ S numbers among $(i + 1)$ groups = ${}^{(m+1-i)+(i+1)-1}C_{(i+1)-1}$ = ${}^{m+1}C_i$, we calculate this for each $i = 0$ to n and sum them up.

For losing:

We can do a similar thing to find the number of ways Faizal can lose:

We iterate for each $i = 1$ to $n + 1$, (we calculate the number of ways Faizal lose when he dies exactly $i + 1$ times, where he dies twice consecutively at the last time)

Ex: If $i = 4$, for Faizal to lose he has to die in the last two rounds consecutively, $_FS_FS_FS_FF$, we calculate the number of ways to distribute $\leq (m - (i - 1))$ S at among i groups = ${}^{(m-i+1)+i}C_i$ = ${}^{m+1}C_i$ and sum them up.

Note: There is a case where Faizal can lose even without consecutively dying twice, we can count that case separately (if $n = 5$, it will be in the form of $_FS_FS_FS_FS_FS_F$) or we can just consider it like like the case where he dies $n + 2$ times $_FS_FS_FS_FS_FS_FF$).

PREREQUISITES:

Combinatorics, Fundamental principle of Counting 5 .

PROBLEM:

Given skill S of N programmers (N is even), we want to find number of ways to form teams such that There doesn't exist two teams $(A, B), (C, D)$ such that $S_D > S_B$ and $S_A > S_C$.

SUPER QUICK EXPLANATION

- Sort the scores in descending (or ascending) order. Try to make pair from left to right while handling programmers having same skill level combined.
- If there are X programmers with same skill, and X is odd, We select the player which will be left out first, and then count the number of ways to make pairs out of remaining players.
- In case we have a left over from higher (or lower) skill yet to be paired, We count first, the number of ways to select one player out of all players of immediately lower (or higher) skill. After that, we do what i mentioned in second point.

EXPLANATION

First of all, As i usually do, consider constraint that no two players have same skill level. We can see that there will always exist exactly one valid team formation, which pairs players like, best with second best, third with fourth and so on till all players are paired.

So, we know, that the multiple valid team formulation arises only when there are multiple players with same skill level.

Now, since this problem involves combinatorics, which i like to explain by examples, so, here we begin examples.

Consider example scores $2, 2, 2, 2$.

We can see that player at position 1 can pair with any of three, and remaining two players pair with each other, giving $3 * 1 = 3$ ways.

Consider example scores $2, 2, 2, 2, 2, 2$.

Same idea, just that first player has 5 choices. Now, we have four players left. Let us assume, we pair the **leftmost unpaired player**, and count number of ways for him to form team. He has 3 choices. After removing him and his partner too, only two players left, which pair with each other, giving $5 * 3 * 1 = 15$ ways.

Consider example scores $2, 2, 2, 2, 2, 2, 2, 2$.

Once again, Same idea, So i ask you to apply this idea to calculate it before seeing its explanation in secret box.

Click to view

Number of choices for leftmost unused player is 7. Six players left.
Number of choice for leftmost unused player is 5 now. Four players left.
Number of choice for leftmost unused player is 3 now. Two players left.
Number of choice for leftmost unused player is 1 now. No players left.
Final number of ways is $7 * 5 * 3 * 1 = 105$ ways.

You all must be wondering why he's boring us with same test case. My point is to ask you all to notice the pattern.

Last example of this type. $2, 2, 2, 2 \ldots 2$ N times.

Now, what did we learn from previous examples. First player has $(N-1)$ choices, next leftmost unused player has $(N-3)$ choice and so on.

Final Number of ways become $(N-1)*(N-3)*\cdots*5*3*1$. This can also be calculated as $\frac{(2*m)!}{2^m*m!}$ where $m = n/2$.

Looking for a proof? Here you go.

When all skill levels are same, the problem is to count number of ways to divide $2*M$ elements into M pairs. We can see, that every permutation of elements correspond to a team formation, where First player is paired with second, third player is paired with fourth and so on. Number of permutations is $(2*M)!$

For example, Take any permutation, say for $M =$ as 1234, this correspond to teams $(1,2),(3,4)$. But, we see, that permutation 1243 also correspond to same teams, as order of players in a team doesn't matter. Team $(1,2)$ is same as $(2,1)$. Since there are M pairs, there are 2^M different permutations representing same teams, just having their players swapped. So, Number of ways of forming team without double counting these swaps is $\frac{(2*M)!}{2^M}$.

But, the ordering of teams is also double counted. Permutation 1234 represent teams $(1,2),(3,4)$ which is same as $(3,4),(1,2)$ represented by permutation 3412.

To avoid double counting, we need to see how many times each team formulation is being counted. It is easy to see that, we have M unordered pairs. Each permutation of these M pairs will be counted. This means that there are $M!$ permutations corresponding to each team, so, we just divide the Number of ways by $M!$.

Final Number of ways become $\frac{(2*M)!}{2^M*M!}$ which gives us correct number of ways to divide $2*M$ players into unordered pairs where permutation of teams doesn't matter. This is what we require.

Till now, we only considered all examples which had same skill level for all players.

Consider another example 1 1 2 2 2 2.

We can see, that players of skill level 1 will be paired among each other, while players of skill level 2 will be paired among each other. By Fundamental principle of counting, we see these are just product of above two. Number of ways for team with skill 1 is 1, while Number of ways to form teams with skill 2 are $3*1$. Total Number of ways is just $1*3=3$ ways.

Now, we know how to count number of teams **When there are even number of players with same skill**.

But when number of players of a skill level is odd, One player of that skill level has to be paired with a player of different skill. Now, **A player cannot be paired with anyone having not immediately lower (or higher) skill.** Player with skill 1 can never be paired with player having skill 3 as long as there exist even a single player having skill 2.

Suppose 1 2 2 3 is the skill levels. Player 1 can be paired with either player 2 or player 3. Other person can be paired with player 4, resulting in 2 total ways.

Our second last example for today is 1 1 1 1 1 2 2 2 2 2 2 3 3 3.

We handle players of same skill together, in increasing (or decreasing, which you like) order of skill.

First we have 5 players of skill 1. We know, One of the player of skill 1 will be paired with player of skill 2. Number of ways to select that 1 player is 5. Remaining 4 players divided into unordered pairs in $3*1$ ways.

Then we have 6 players of skill 2 and one player of skill 1 left. Firstly, we can pair player with player 1 with any of the six, In six ways. Now, 5 players of skill level left. One chosen player will be left out, we can choose in 5 ways. 4

players left, which are divided into pairs in $3 * 1$ ways.

Lastly, we have 3 players of skill 3 and one player of skill 2 to be paired. Once again, we choose player to be paired with player of skill 2, which can be done in 3 ways. Two players of skill 3 left, which can be divided into teams in 1 way.

Since all these events have to be performed together, then by principle of counting, the final answer is just product of all these. that is. Required Number of ways is $5 * 3 * 6 * 5 * 3 * 3 * 1 = 4050$ ways which is the required answer.

This was all for today, though you can read more about the proof here 5 , as well as, you can find proof by searching about "Dividin N elements into pairs."

Bonus Question

Instead of SnackDown, Given skill of N players, Make teams for ACM-ICPC regionals. It is given that N is divisible by 3. Leave answer in comments.

Time Complexity

Time complexity is $O(N * LogN)$ due to sorting. Ordered Frequency map can also be used and is preferable.

PS: The solutions based on same idea, which tried to make a frequency array instead of frequency map, got TLE, because of the test file can have up to 1000 test cases of small size, in which case, your solution tried to complete $O(T * 10^6)$ iterations which will surely time out.

PREREQUISITES:

Number theory, Inclusion-Exclusion principle, Combinatorics

PROBLEM:

We define the *GCD set* of an array A of length N as the set containing the GCD of all prefixes of A. More formally, the GCD set G of A contains the values $\gcd([A_1]), \gcd([A_1, A_2]), \gcd([A_1, A_2, A_3]), \ldots, \gcd([A_1, A_2, \ldots, A_n])$, where $\gcd(B)$ is the largest integer d such that all the numbers in array B are divisible by d. Note that **all values in G are distinct**.

For example, $\gcd([6]) = 6$, $\gcd([6, 12]) = 6$, $\gcd([6, 12, 9]) = 3$, and $\gcd([6, 12, 9, 2]) = 1$. Therefore, the GCD set of the array $[6, 12, 9, 2]$ is $\{1, 3, 6\}$.

You are given an integer set S of size M and two integers N, MOD, where MOD is a prime. You need to calculate the probability that the GCD set of P is equal to S, where P is a **permutation** of values 1 to N chosen uniformly randomly.

It can be shown that the answer is in the form of $\frac{P}{Q}$ where P and Q are coprime non-negative integers and $Q \not\equiv 0$. You need to find the value $P \cdot Q^{-1} \pmod{MOD}$.

EXPLANATION

During the calculation of the gcd-s in the following order:
$\gcd([A_1]), gcd([A_1, A_2]), \gcd([A_1, A_2, A_3]), \ldots, \gcd([A_1, A_2, \ldots, A_n])$ we can observe few properties:

- In every step the result cannot exceed the previous one, moreover
- In every step the result is always a divisor of the previous one because $gcd([a, b, c]) = gcd(gcd(a, b), c)$
- At the last step gcd result is 1: because the 1 occurs in the array
- $A_1 = S_M$

We also can realize if a permutation generate S set then the first item of the permutation is S_M. It seems to calculate the next item of a good permutation we have to calculate how much integer exists in the range $1 \le x \le N$ to get the second item of the set: $gcd(S_M, x) = S_{M-1}$ or more generally:

Count the number of $1 \le x \le N$ where $gcd(a, x) = b$.

This equivalent with the following equation: $1 \le x \le N/b$ where $gcd(a/b, x) = 1$. So firstly we should count a relative primes to a given number c in a range $[1, K]$.

Its easy to see its enough to solve the special case when c is a product of different prime numbers, because if a number is relative prime to a prime number then it is also relative prime any power of that prime number.
First consider a simple case when c is a prime then the non-relative primes in the range : $c, 2 \cdot c, .., (K/c) \cdot c$, so the number of solution is $K - K/c$. In the case c is product of two different prime numbers $p_1 \times p_2$ then the number of non-relative numbers $K - K/p_1$ + $K - K/p_2$ but we subtract in both the following numbers: $p_1 \cdot p_2, 2 \cdot p_1 \cdot p_2, ..., \frac{K}{c} \cdot p_1 \cdot p_2$ so we get the number of relative primes $K - \frac{K}{p_1} - \frac{K}{p_2} + \frac{K}{p_1 \cdot p_2}$. We can solve it generally when c is a product of different prime factors with using the [inclusion-exclusion principle] ([Inclusion–exclusion principle - Wikipedia]). Because of the N is less than 1 billion, the number of different prime factors cannot exceed 9, so we can use the above principle it has at most 2^9 term.

Lets consider the following partition of the $[1, N]$ range: $C_1, C_2, .., C_M$ where :

- $C_M = \{S_M\}$
- $C_{M-1} = \{x : gcd(x, S_M) = S_{M-1}\}$
- $\dots$
- $C_i = \{x : gcd(x, S_{i+1}) = S_1\}$

If we want to create a permutation what generates the S set then we have to produce the gcd's in decreasing order so lets consider the case when the last generated gcd value is S_i, and we want to change gcd to S_{i-1}. In this case we have to choose a number from C_{i-1}.

- If we choose a number from a larger index partition then gcd won't change
- If we choose a number from a smaller index partition then we get smaller gcd than S_{i-1}, after that we cannot generate the S set.

With these observations we can define the a permutation is good if and only if the first appearance of an item from C_i then all at least one item from the C_j where $j > i$ already appeared, and no item occured from C_k where $k < i$.

Calculate the final probability from the C_i's : If we consider the N slots, firstly we have $|C_M|$ option. The remaining integers from C_M we can assign any slots so we have $(N-1) \cdot (N-2) \cdot ... \cdot (N - |C_M|)$ possibility. Now check the C_{M-1} set: the first item has to go to the first non used slot, we have |C_M-1| choice which item put there. The other items can assign all the non used slots so we have $(N-|C_M|-1)\cdot (N-|C_M|-2) \cdot …\cdot(N-|C_M|-|C_{M-1}|) options.

So we get the final number of correct permutation is : $\frac{N! \cdot \prod_{i=1}^{K} C_i}{\prod_{i=1}^{K} (\sum_{j=1}^{i} C_j)}$. Because of the task asks probability of the correct permutation we can get rid off the $N!$ term :

$$\frac{\prod_{i=1}^{K} C_i}{\prod_{i=1}^{K} (\sum_{j=1}^{i} C_j)}$$

TIME COMPLEXITY:

$O(sqrt(N) + M)$ per test case

PREREQUISITES:

Combinatorics , FFT/NTT

PROBLEM:

Given an array of size N , and two integer X and Y . You have to answer Q queries, for each query you have to find how many ways to form a subsequence of size K of the original array such that the number of elements in the subsequence which are divisible by X is divisible by Y.

QUICK EXPLANATION:

Let's count the number of elements that are divisible by X, and call this number d.
Then there will be $N - d$ elements that are not divisible by X.

Now for a given K:

let S be set of all the numbers less than or equal to $\leq K$ which are divisible by X.

for example if X=7 and K=35
X = {0 , 7 , 14 , 21 , 28 , 35}

We need to find the number of ways to simultaneously select X_i distinct elements from d distinct elements and $K - X_i$ distinct elements from $N - d$ distinct elements, and take sum for all the values in the set X.

It will be of the form:

${}^{d}C_{X_1} \,.\, {}^{N-d}C_{k-X_1} + {}^{d}C_{X_2} \,.\, {}^{N-d}C_{k-X_2} + \ldots$

which in turn is coefficient of x^k in the product of polynomials

1. ${}^{d}C_{X_1} \,.\, x^{X_1} + {}^{d}C_{X_2} \,.\, x^{X_2} + \ldots + {}^{d}C_{X_i} . x^{X_i} + \ldots$

2. ${}^{N-d}C_0 \,.\, x^0 + {}^{N-d}C_1 \,.\, x^1 + \ldots + {}^{N-d}C_{N-d}$

Brute Force way to multiply polynomial takes $O(n^2)$ time, but we can use FFT to do the polynomial multiplication in $O(nlogn)$ time.
Then we can answer each query in $O(1)$ time.

PREREQUISITES:

Dynamic programming or combinatorics.

PROBLEM:

Dehatti wonders how many ways can his team score exactly R runs in B balls with L wickets. Given that each ball can result in $4, 6, 0$ runs or a wicket(wicket won't add anything to score).

QUICK EXPLANATION:

Suppose we want to make R runs in B balls, having X number of 4's, Y number of 6's, $(W \leq L)$ number of wickets and Z number of 0's. So there will be $4X + 6Y = R$ and $X + Y + W + Z = B$. If (x, y, z, w) is a particular solution to given equations, then number of ways of arrangement will be $\frac{B!}{(x!)(y!)(w!)(z!)}$. We have to add all such cases, and the given time limit is sufficient for it.

EXPLANATION:

First of all, pre-calculate the value of ${}^nC_r$ which can be done using a simple dynamic programming approach. You may refer this link 9 for the more details.

As batsman has to make R runs in B balls, if $R > 6B$, there will not be any way of doing it.

Suppose he makes R runs in B balls, with at-max L wickets. So, we will have the following equations.

$$4 \times \text{fours} + 6 \times \text{sixes} = R$$
$$\text{fours} + \text{sixes} + \text{wickets}(\leq L) + \text{zeros} = B$$

There will be many solutions to the given equations for particular values of R,B and L. We have to consider each one of them.

Let us consider that a particular solution of the equation is fours = X, sixes = Y, wickets = W and zeros = Z, or (X, Y, Z, W) is a solution to the given equations.

So, number of ways of arranging X $4's$, Y $6's$, W wickets ans Z $0's$ can be calculated easily using the formula.

$$\text{Ways} = \frac{B!}{(X!)(Y!)(Z!)(W!)}$$
$$= \binom{B}{X}\binom{B-X}{Y}\binom{B-X-Y}{W}$$

Values can be used from initially calculated $\binom{n}{r}$ values . Take care of modulus as well.

T only thing remains is to produce all the triplets (X, Y, Z, W).

We can run a loop on number of sixes varying from 0 to min(B,R/6),number of fours will be fixed i.e.

```
Number of fours = (R -6*sixes)/4 if ((R - 6*sixes) % 4 == 0)
```

And we can loop over number of wickets from 0 to L.

```
Number of zeros = B-sixes-fours-wickets
```

The following piece of self explanatory code will do all the calculations.

```
for(int six=0; six*6 <= r && six <= b; six++) {
        int other = r - six*6;
        if(other % 4 == 0) {
            int four = other/4;
            for(int wicket=0; wicket <= l; wicket++) {
                if(six + four + wicket <= b) {
                    long long cur = C[b][six];
                    ( cur *= C[b-six][four] ) %= 1000000007;
                    ( cur *= C[b-six-four][wicket] ) %= 1000000007;
                    ways += cur;
                }
            }
            ways %= 1000000007;
        }
    }
```

Solutions

PREREQUISITES:

Rule of product in combinatorics 38 , Modulo operation 11

PROBLEM:

You are given 3 positive integers N_1, N_2, N_3 up to 10^{18} and need to count the number of triples (X_1, X_2, X_3) such that $1 \le X_i \le N_i$ for $i = 1, 2, 3$. You should output the answer modulo $10^9 + 7$.

QUICK EXPLANATION:

By sorting input numbers we can assume that $N_1 \le N_2 \le N_3$. Then the answer is $N_1 * (N_2 - 1) * (N_3 - 2)$. Watch out for integer overflows during calculation of this number modulo $10^9 + 7$.

EXPLANATION:

If we sort numbers N_1, N_2, N_3 the answer remains the same. Hence let's assume that $N_1 \le N_2 \le N_3$. Then for X_1 we have N_1 choices, for X_2 we have $N_2 - 1$ choices and for X_3 we have $N_3 - 2$ choices. Indeed:

1. X_1 can be any number from 1 to N_1, inclusive. There are N_1 such numbers 🙂
2. X_2 can be any number from 1 to N_2, inclusive, which is not equal to X_1. Since $X_1 \le N_1 \le N_2$, there are $N_2 - 1$ such numbers (numbers from 1 to N_2, inclusive, with excluded X_1). Clearly, if $N_1 = N_2 = 1$ then the answer is zero since the only choice for X_1 and for X_2 is 1, so any such triple will have equal numbers.
3. Finally, if $N_3 \ge 2$ then X_3 can be any number from 1 to N_3, inclusive, which is not equal to X_1 and X_2. Since $X_1 \le N_1 \le N_3$, $X_2 \le N_2 \le N_3$ and $X_1 \ne X_2$, there are $N_3 - 2$ such numbers (numbers from 1 to N_3, inclusive, with excluded X_1 and X_2).

Now using rule of product 38 in combinatorics we get that the answer is simply the product of N_1, $N_2 - 1$ and $N_3 - 2$ as mentioned above. Note that when $N_2 = 1$ this formula also works so we don't need to handle this case separately.

Probably the trickiest part is to output the answer modulo $M = 10^9 + 7$ having numbers N_i up to 10^{18}. There are several things you should notice in order to get AC:

1. You should use 64-bit integer type to store and input numbers N_i.
2. When you want to multiply two numbers up to 10^{18} modulo $10^9 + 7$ you should at first take them modulo $10^9 + 7$ and then do the multiplication using 64-bit integer type. Otherwise the result will be wrong due to overflow. Even such code is wrong
   ```
   A % M * B % M
   ```
   since you will multiply numbers **A % M** and **B** at the second step and this will lead to overflow (**A % M** is about **1e9** and **B** is about **1e18**, so their product is about **1e27** which does not fit in 64-bit integer type). Such overflow will occur on almost every random test.
3. Another common mistake is to do like that
   ```
   A = N1 % M
   B = (N2 - 1) % M
   C = (N3 - 2) % M
   ans = A * B * C % M
   ```
   Here at the final step you multiply three numbers up to about **1e9**. Their product could be up to **1e27** which does not fit in 64-bit signed integer. Therefore integer overflow occurs. The correct code would be
   ```
   ans = A * B % M * C % M
   ```
   Here we take the product **A * B** modulo **M** so it becomes smaller than **M** and at the final step you multiply two numbers up to about **1e9**. Their product is up to about **1e18** and fits in 64-bit integer type.

4. You can't replace numbers $\mathbf{N_i}$ by their residues modulo $\mathbf{10^9 + 7}$ before sorting. Indeed, the order of numbers can change after that and you get completely another number as the answer. The reference test is: `2 3 1000000008`. Actually it happens often on any large random test.

5. If after sorting you replace numbers $\mathbf{N_i}$ by their residues modulo $\mathbf{10^9 + 7}$ and then calculate the answer as
   `ans = N1 * (N2 - 1) % M * (N3 - 2) % M`
   then your answer could be negative in the end. The reference test is: `1 2 1000000008`.
   Hence if you are using this method make the check
   `if ans < 0 then ans = ans + M`
   after that.

6. As mentioned one of the ways how to do all properly is to calculate answer as follows
   `ans = (N1 % M) * ((N2 - 1) % M) % M * ((N3 - 2) % M) % M`
   Note that at first sight this way also could lead to negative answer. But this does not happen here. See comments in tester's solution 152 for explanation.

Actually, it is easy to see that this solution could be generalized to any number of variables instead of 3. Namely, if we have numbers **N[0], N[1], …, N[K-1]** and want to count the number of arrays **(X[0], X[1], …, X[K-1])** such that **1 ≤ X[i] ≤ N[i]** for **i = 0, 1, …, K-1**, and all **X[i]** are different, then after sorting numbers **N[i]** the answer will be the product of **N[i] − i** for **i = 0, 1, …, K-1**. See tester's solution 152 as a reference.

ALTERNATIVE SOLUTION:

The problem could be solved using inclusion-exclusion principle. The formula for the answer in this case is
$\mathbf{N_1 * N_2 * N_3 - \min\{N_1, N_2\} * N_3 - \min\{N_2, N_3\} * N_1 - \min\{N_3, N_1\} * N_2 + 2 * \min\{N_1, N_2, N_3\}}$.
See author's solution 173 as a reference. It also contains explanatory comments to this formula.

PREREQUISITES:

Maths, Combinatorics

PROBLEM:

You are given an array A consisting of N integers and Q queries. Each query is described by two integers L and R. For each query, output the number of tuples (i, j, k) such that $L \leq i < j < k \leq R$ and $A_i + A_j + A_k$ is an even number.

EXPLANATION:

Since we are looking for even tuples, let us see when will the combination of three numbers result in an even sum. We can easily see that the conditions to get an even sum are:

$$Even + Even + Even = Even$$
$$Odd + Odd + Even = Even$$

Let's say E and O are the count of even and odd numbers respectively within the queried range of positions, $[L, R]$ in the given array.

Calculating E and O in a queried window of A

Running a loop from positions L to R for every query would result in a complexity of $O(N \times Q)$ for calculation of even and odd numbers in the queried window of the array.

Instead, we precompute the number of even numbers from position 1 to x for every $x \in [1, N]$ and store them in an array (say B), i.e. $B_x =$ number of even numbers in $A_{[1,x]}$. This makes it clear that the number of even numbers, E in $A_{[L,R]}$ will be given by B_R when $L = 1$ and by $B_R - B_{L-1}$ otherwise. Similarly the number of odd numbers, O in $A_{[L,R]}$ will be given by $(R - L + 1) - E$.

As in the first type of tuples, we need to choose 3 even numbers from the total even numbers present in the queried window of the given array, the total number of tuples of the first type are:

$$fst = \binom{E}{3}$$

Now in the second type of tuples, we need to choose 2 odd numbers from the total odd numbers along with 1 even number among all even numbers present in $[L, R]$. Hence the total number of tuples of the second type will be:

$$snd = \binom{O}{2} \times \binom{E}{1}$$

Thus the total number of tuples with even sum in the array window $[L, R]$ would become:

$$total = fst + snd$$

That is our final answer for every query, we can simply output it.

TIME COMPLEXITY:

$O(N + Q)$ per test case

PREREQUISITES:

Inclusion-Exclusion Principle, Counting

PROBLEM:

Given a simple undirected graph. Count the sets of edges, which when added to the graph make it 2-regular.

EXPLANATION:

If the graph has any vertex of degree > 2, the answer is clearly 0. Otherwise, all degrees are ≤ 2 and the graph consists of singletons, chains and cycles.

Subtask 1

We have $K = N(N-1)/2 - M$ edges that can be added to the graph. Consider any ordering $e_1, e_2,, \ldots e_K$ of these edges. The state of the graph can be represented a degree vector. So there are 3^n valid states (for each node, degree 0, 1 or 2) and we can define a simple recursion based on whether we add e_i or not. The time complexity would be $O(N^2 3^N)$.

Subtask 2 and 3

Let there be a singletons, b chains of size 2 and c chains of size ≥ 3 in the graph.

Define $F(a, b, c)$ to be the number of sequences of edges we can add to a graph with a singletons, b chains of size 2 and c chains of size ≥ 3 to make it 2-regular. Consider the first added edge. The following are the possibilities:

- The edge connects two endpoints of a single chain : $c \times F(a, b, c-1)$
- Two singletons : $\binom{a}{2} \times F(a-2, b+1, c)$
- Two chains of size 2 : $4 \times \binom{b}{2} \times F(a, b-2, c+1)$
- Two chains of size ≥ 3 : $4 \times \binom{c}{2} \times F(a, b, c-1)$
- A singleton and a chain of size 2 : $2 \times ab \times F(a-1, b-1, c+1)$
- A singleton and a chain of size ≥ 3 : $2 \times ac \times F(a-1, b, c)$
- A chain of size 2 and a chain of size ≥ 3 : $4 \times bc \times F(a, b-1, c)$

In $F(a, b, c)$ we have considered every valid edge set $(a+b+c)!$ times, as the edges can be added in any order. So, the final answer is $F(a, b, c)/(a+b+c)!$

The time complexity here is $O(N^3)$

Code

```
int val(int A, int B, int C){
    if(A + B + C == 0) return 1;
    if(A + B + C == 1) return C == 1;
    for(int a = 0; a <= A + B + C; a++)
        for(int b = 0; a + b <= A + B + C; b++)
            dp[a][b] = 0;
    dp[0][0] = 1;
    for(int s = 2; s <= A + B + C; s++){
```

```
    for(int a = 0; a < s; a++){
        for(int b = 0; a + b < s; b++){
            dp2[a][b] = dp[a][b];
            dp[a][b] = 0;
        }
    }
    // a <= A
    int mx_a = min(s, A);
    for(int a = 0; a <= mx_a; a++){
        int mx_b = min(B + A / 2, s - a);
        for(int b = 0; b <= mx_b; b++){
            ll ret = 0;
            int c = s - a - b;
            if(c) ret += c * 1LL * dp2[a][b];

            if(a >= 2) ret += C2(a) * dp2[a - 2][b + 1];

            if(b >= 2) ret += 4LL * C2(b) * dp2[a][b - 2];

            if(c >= 2) ret += 4LL * C2(c) * dp2[a][b];
```

Subtask 4

We are basically trying to make cycles from the chains in the original graph. Consider a singleton as a chain of siz 1. Then in each cycle, the sum of the sizes of the chains used has to be ≥ 3.

Consider those ways in which we end up with exactly k cycles. For each cycle, there are two ways of giving it a direction (clockwise or anticlockwise). So, we will consider directed cycles and divide by 2^k. Each ways of choosing directed cycles represents a permutation of the chains. We have to exclude cycles of 3 kinds in this permutation:

1. Those consisting of two chains of size 1
2. Those consisting a single chain of size 1
3. Those consisting of a single chain of size 2.

For this, we will use inclusion exclusion. Let's say we are excluding q particular cycles of the first kind. The number of ways to choose these is $\binom{a}{2q} \times (2q-1)!!$, as we can choose $2q$ individual chains in $\binom{a}{2q}$ ways, and then group them in $(2q-1)!!$ ways. After this, note that the chains of the second and third kind are similar (the only restriction is that they can't be mapped to themselves). If there are a total p chains of second and third kind we're excluding, we can choose them in $\binom{a+b-2q}{p}$ ways. For the remaining $a+b+c-p-2q$ chains, we can have any permutation. If the remaining part had k cycles, we need to multiply by $\dfrac{1}{2^{p+q+k}}$ as there are $p+q+k$ total cycles. Also, we need to multiply by $(-1)^{p+q}$ for the inclusion exclusion part.

Let $F(n)$ be the sum of $2^{-\text{number of cycles}}$ over all permutations of n elements.

Then, $F(n) = \dfrac{F(n-1)}{2} + (n-1)f(n-1) = (n-\frac{1}{2})F(n-1)$.

This is because, we can either map n to itself, increasing the number of cycles by 1 or we can map it to some of the other $n-1$ elements, keeping the number of cycles the same.

Then, our answer can be written as:

$$\sum_{q=0}^{a/2} \sum_{p=0}^{a+b-2q} (-1/2)^{p+q} \binom{a}{2q} \binom{a+b-2q}{p} (2q-1)!!\ F(a+b+c-p-2q)$$

After precomputing factorials, double factorials and all the F values in $O(N)$, the answer can be easily be found in $O(N^2)$ by iterating on p and q.

Subtask 5

Let's expand $\binom{a+b-2q}{p}$ in the above expression. Then the summand can be expressed as $u_p \times v_q \times w_{p+2q}$, where,

- $u_p = (-1/2)^p \frac{x^p}{p!}$
- $v_q = (-1/2)^q \binom{a}{2q} (2q-1)!!(a+b-2q)!$
- $w_r = \frac{F(a+b+c-r)}{(a+b-r)!}$

Consider $U(x) = \sum_{p=0}^{a+b} u_p x^p$ and $V(x) = \sum_{q=0}^{a/2} v_q x^{2q}$

Let c_r be the coefficient of x^r in $A(x)B(x)$,

Then our answer is $\sum_{r=0}^{a+b} c_r w_r$

The time complexity is $O(N \log(N))$

PROBLEM

Given a string S consisting only of lowercase english characters, how many strings M of length N are possible such that S can be obtained by removing a substring from M.

EXPLANATION

It is obvious that if N< = M.size() the ans will be 0 but if N>M.size(),we can derive a relation as follows.

If we put the whole string M at the beginning, we can put power(26,N-M.size()) characters at the end of the string.

Now, we also need to try to put M.size()-i characters at the beginning and i characters at the end for all 1 <= i <= M.size().

Consider a case where M="abc" and N=5. Then for i=1 put 2 characters of string M at the beginning and 1 character of string M at the end so the new string will be "ab c".However , we previously calculated this case "abc " , so now we need to calculate only "ab c". If we put ‘c’ in the first space (third letter) we will have duplicates because we already calculated all strings that starts by "abc" so in order to ensure uniqueness, we need to let the first space be any character other than ‘c’ and for rest of the characters can be any possible character, so the number of ways will be 25 for first space multiplied by power(26,N-M.size()-1) for the rest of spaces.

This can be done for each i from 1 to M.size() so the answer for all i will be M.size() * 25 * power(26,N-M.size()-1)

The final answer will then be power(26,N-M.size()+M.size()*25*power(26,N-M.size()-1).

PREREQUISITES:

Maths, Combinatorics, Prefix sum

PROBLEM:

Given an array of digits from 0 - 9. You have to count total number of evil subarrays of the given array. A subarray is evil if sum of all elements in that subarray is equal to the size of the subarray.

QUICK EXPLANATION:

For subtask 1 we can easily do brute force using two nested loops and overall time complexity would be $O(T * M^2)$.

For subtask 2, Store the frequency of $A_i - i$ for $(0 \leq i < M)$. Each distinct key will contribute $\binom{n}{2} = \frac{n*(n-1)}{2}$ to the total count where n is the frequency of the key.

EXPLANATION:

First of all we precalculate a prefix sum array P, where $P_i = \sum_{j=0}^{i-1} A_i$. It means P_x is sum of first x elements of array A.

Let's assume $[l, r]$ is an evil subarray, then
$P_r - P_{l-1} = r - l + 1$
$\Rightarrow P_r - P_{l-1} = r - (l - 1)$
$\Rightarrow P_r - r = P_{l-1} - (l - 1)$

From here we can say that if there are K such i whose $P_i - i$ value is same then selecting any two i from it, always results in an evil subarray. So we just need to store the frequency of all the distinct values of $P_i - i$ (can be stored using a hash map) and for each frequency x we have to add $\frac{x*(x-1)}{2}$ to the answer.

Overall time - complexity of this approach is $O(T * M)$.

PREREQUISITES

Combinatorics, Pointers, and Patience.

PROBLEM

Given a matrix A with N rows and M columns and an integer X. Let $f(X)$ denotes the number of submatrices B inside A such that $min(B) \bigoplus max(B) = X$.

Find the sum of $f(X)$ over all permutations of rows of A, and compute their sum.

QUICK EXPLANATION

EXPLANATION

From permutations to subsets

Assume rows and columns are 1-indexed. Considering a submatrix B, it shall consist of a range $[L, R], 1 \leq L \leq R \leq M$ of columns of rows U to B in some permutation of rows. We can see that it shall correspond to some subset of rows (not necessarily continuous) in the original matrix A. Let S denote the set of rows included in the subset. We shall count the number of permutations in which rows present in this subset appear together in any order (only then they can be chosen as rows of a submatrix).

Let's assume C denotes the number of rows in subset. We don't care about the order of rows in subset, so there are $C!$ orderings of rows within subset. Now we can consider all this set of rows as a single row. So there are $N - C + 1$ rows which can be ordered freely. Hence, the number of permutations of rows, in which subset S appears as continuous set of rows is $C! * (N - C + 1)!$

Let's assume, for a fixed set of rows S, the number of submatrices consisting of elements of these rows is W_S and $min(B) \bigoplus max(B) = X$, then we need to find $\sum\limits_{S \subseteq P(R)} W_S * (|S|!) * (N - |S| + 1)!$, where $P(R)$ denotes the power set of rows. Also, $W_S = 0$ if $S = \emptyset$

Computing W_S

Let's consider a naive way. For a subset S, each submatrix correspond to a continuous set of columns, say from L to R for $1 \leq L \leq R \leq M$. Trying all pairs, we can count the number of pairs (L, R) such that the submatrix formed by columns L to R of subset of rows S has $min(B) \bigoplus max(B) = X$. This solution works in $O(2^N * M^2)$ and shall time out.

Code

```
import java.util.*;
import java.io.*;
class MAXXMIN{
    //SOLUTION BEGIN
    void pre() throws Exception{}
    void solve(int TC) throws Exception{
        int N = ni(), M = ni(), X = ni();
        long[] fact = new long[1+N];
```

```
        fact[0] = 1;
        for(int i = 1; i<= N; i++)fact[i] = fact[i-1]*i;
        int[][] A = new int[N][M];
        for(int i = 0; i< N; i++)
            for(int j = 0; j< M; j++)
                A[i][j] = ni();

        long ans = 0;
        for(int mask = 1; mask < 1<<N; mask++){
            int[] imin = new int[M], imax = new int[M];
            Arrays.fill(imin, Integer.MAX_VALUE);
            Arrays.fill(imax, Integer.MIN_VALUE);
            for(int r = 0; r< N; r++){
                if(((mask>>r)&1) == 1){
                    for(int c = 0; c< M; c++){
                        imin[c] = Math.min(imin[c], A[r][c]);
                        imax[c] = Math.max(imax[c], A[r][c]);
                    }
                }
            }
            long subarrays = 0;
```

Computing W_S in $O(M * logM)$

Let $min_c = \min_{r \in S} A_{r,c}$ and $max_c = \max_{r \in S} A_{r,c}$. Now we need to compute pairs (L, R) such that $\min_{c=L}^{R} min_c \oplus \max_{c=L}^{R} max_c = X$.

Let's try divide and conquer here. For range $[L, R]$, if p denotes any position of $\min_{c=L}^{R} min_c$, then all intervals (l, r) such that $L \le l \le p \le r \le R$ shall have $\min_{c=L}^{R} min_c = min_p$. With fixed minimum, we need to count the pairs (l, r) such that $L \le l \le p \le r \le R$ such that $\max_{c=l}^{r} max_c = min_p \oplus X$.

Since $min_p \oplus X$ is a fixed value, we can run binary searches now. Let's find largest position L_g such that $max_{L_g} > min_p \oplus X$, position L_{ge} such that $max_{L_{ge}} \ge min_p \oplus X$ and smallest positions R_g such that $max_{R_g} > min_p \oplus X$ and position R_{ge} such that $max_{R_g} \ge min_p \oplus X$. Note that if $L_g < L$ or $L_{ge} < L$, we can use $L - 1$. Similarly $R + 1$ for right ends.

We need subarrays with left endpoint in range $[L, p]$ and right end in range $[p, R]$ such that maximum of range $[l, r]$ is $min_p \oplus X$. We can see that the number of such subarrays is given by $(p - L_g) * (R_g - p) - (p - L_{ge}) * (R_{ge} - p)$. $(p - L_g) * (R_g - p)$ denote the subarrays with maximum $\le min_p \oplus X$ and $(p - L_{ge}) * (R_{ge} - p)$ denotes the number of subarrays with maximum $< min_p \oplus X$.

This way, by buildning RMQ on both min and max arrays, we can solve the problem in $O(M * log(M))$ for each set S, leading to time complexity $O(2^N * M * log(M))$ per test, which is yet too slow to get AC. Some optimizations might be able to AC.

Code

```
import java.util.*;
import java.io.*;
```

```
class MAXXMIN{
    //SOLUTION BEGIN
    void pre() throws Exception{}
    void solve(int TC) throws Exception{
        int N = ni(), M = ni(), X = ni();
        long[] fact = new long[1+N];
        fact[0] = 1;
        for(int i = 1; i<= N; i++)fact[i] = fact[i-1]*i;
        int[][] A = new int[N][M];
        int[] all = new int[N*M];
        for(int i = 0; i< N; i++)
            for(int j = 0; j< M; j++)
                all[i*M+j] = A[i][j] = ni();

        Arrays.sort(all);
        int C = 1;
        for(int i = 1; i< all.length; i++)if(all[i] != all[C-1])all[C++] = all[i];
        all = Arrays.copyOf(all, C);
        for(int i = 0; i< N; i++)
            for(int j = 0; j< M; j++)
                A[i][j] = Arrays.binarySearch(all, A[i][j]);

        int[] nxt = new int[C];
        for(int i = 0; i< C; i++){
            int pos = Arrays.binarySearch(all, all[i]^X);
            if(pos >= 0)nxt[i] = pos;
```

$W_S : O(M)$

We can no longer build RMQ, so let's compute previous and next smaller, and previous and next greater elements. Also, for position c, let's compute smallest position $p \geq c$ such that $max_p = min_c \bigoplus X$. and let's compute largest $p \leq c$ such that $max_p = min_c \bigoplus X$

Our aim is still the same, to consider all positions p one by one, find the interval in which position p is minimum(say $[L, R]$), and among subarrays (l, r) such that $L \leq l \leq p \leq r \leq R$, find the number of subarrays with maximum $min_p \bigoplus X$ by computing L_g, L_{ge}, R_g, R_{ge} using these arrays. I have added comments in my code for better understanding.

Implementation note: While having multiple ocurrences of min_p, be sure not to doublecount.

TIME COMPLEXITY

The time complexity is $O(2^N * M)$ per test case.

PROBLEM:

Given an array *A* of *N* integers, a position *x*, and a number *M*, find the value of A_x after performing the following operation *M* times:

```
for i = 2 to N:
    A[i] = A[i] + A[i-1]
```

As the answer could be large, the answer should be given modulo $1\,000\,000\,007$.

EXPLANATION:

Subtask1

In the first subtask, we have $x \leq 2$. So we only have 2 cases:

- $x = 1$: A_1 is never changed in the given procedure, so we already have the result (don't forget the modulo!)
- $x = 2$: in each execution of the above procedure, we add A_1 once to A_2. So the result is just $A_2 + M \times A_1$.

Subtask 2

The second subtask guarantees that $N \times M \leq 10^6$. This means that we can just perform the given algorithm *M* times. This approach takes $O(N \times M)$ time, which is ok with these constraints.

Subtask 3

The number N is irrelevant if larger than x because A_x is only affected by indices smaller than x. Hence, I will consider $N = x$ for the rest of the editorial, i.e., we will want to know the value of A_N.

When we are given a set of operations to be performed, a good starting point is to see what happens after a few iterations. Suppose we have $N = 5$, and the initial values are $A_1 = a$, $A_2 = b$, $A_3 = c$, $A_4 = d$, and $A_5 = e$.

Step	A_1	A_2	A_3	A_4	A_5
start	a	b	c	d	e
1	a	a + b	a + b + c	a + b + c + d	a + b + c + d + e
2	a	2a + b	3a + 2b + c	4a + 3b + 2c + d	5a + 4b + 3c + 2d + e
3	a	3a + b	6a + 3b + c	10a + 6b + 3c + d	15a + 10b + 6c + 3d + e
4	a	4a + b	10a + 4b + c	20a + 10b + 4c + d	35a + 20b + 10c + 4d + e
5	a	5a + b	15a + 5b + c	35a + 15b + 5c + d	70a + 35b + 15c + 5d + e

You may recognise that the factors associated with each initial value are binomial coefficients 14 .

1								
1	1							
1	2	1						
1	3	3	1					
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1

In each step, we are summing the values of a column in the Pascal Triangle. In the above example, $M = 5$, so we are interested in summing the first 5 lines in each column. For example, the value associated with *a* in the final A_5 is the result of $\binom{3}{3} + \binom{4}{3} + \binom{5}{3} + \binom{6}{3} + \binom{7}{3}$.

It can be shown (see here 30) that $\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$. In fact, $\sum_{m=0}^{7} \binom{m}{3} = \binom{8}{4} = 70$.

So, we now have a formula to get the coefficient of each initial value, in order to get the final A_N. In the above example, we have $A_5 = \binom{M+3}{4}a + \binom{M+2}{3}b + \binom{M+1}{2}c + \binom{M}{1}d + \binom{M-1}{0}e =$
$\binom{8}{4}a + \binom{7}{3}b + \binom{6}{2}c + \binom{5}{1}d + \binom{4}{0}e =$
$70a + 35b + 15c + 5d + e$

The binomial coefficient can be calculated as $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. M is up to 10^{18} so it is too big to calculate the factorial directly. However, we can take advantage of the formula we are calculating.

$\binom{M+3}{4} = \frac{(M+3)!}{4!(M+3-4)!} = \frac{(M+3)!}{4!(M-1)!} = \frac{M*(M+1)*(M+2)*(M+3)}{4!}$

$\binom{M+2}{3} = \frac{M*(M+1)*(M+2)}{3!}$

$\binom{M+1}{2} = \frac{M*(M+1)}{2!}$

$\binom{M}{1} = \frac{M}{1!}$

$\binom{M-1}{0} = 1$

As you can see, we can calculate each factor iteratively using the previous value. In order to divide by each factorial, we need to use the Modular Multiplicative Inverse 18 , which can be computed in $O(\log N)$. So, the time complexity of this approach is $O(N \log N)$.

PREREQUISITES:

Basic Combinatorics, Probability, Modular Multiplicative Inverse 7

PROBLEM:

Given the length L_i of string which may contain both uppercase and lowercase English alphabets, find the probability that this string will be a palindrome, under the condition that for exactly K $(0 \leq K \leq 26)$ alphabets, their uppercase and lowercase representations are treated to be similar.
This probability can be represented as P/Q where P and Q are integers. Thus, print the result as $P.Q^{-1} \mod 10^9+7$, where Q^{-1} denotes the modular multiplicative inverse of Q(modulo (10^9+7)).

EXPLANATION:

Considering all 52 (uppercase and lowercase) characters, it can be proved that the total number of strings of length L will be 52^L, since each one of the L positions in the string can be occupied by any of the available 52 characters.

For any string S of length L to be a palindrome, $S[i] = S[L-i]$ must be satisfied, $\forall\ i \in [1, L]$. Thus for every pair of indices $(i, L-i)$, we need similar characters at both indices.

- Here, the order of indices in pairs do not matter, i.e. $(i, L-i)$ will be same as $(L-i, i)$.
- Also, for odd L this condition will always hold true for center element (i.e. for $(\frac{L}{2}, \frac{L}{2})$).

Thus we have total $\lfloor \frac{L}{2} \rfloor$ such pairs to deal with.

For $K = 0$:

If $K = 0$, then for each pair we need to have exactly same character at both the indices.
Since we can choose any of the 52 characters for any of the $\lfloor \frac{L}{2} \rfloor$ pairs, total number of palindromes will be $52^{\lfloor \frac{L}{2} \rfloor}$ in this case.

Hence, in this case, the numerator will be $52^{\lfloor \frac{L}{2} \rfloor}$ and denominator will be 52^L, after performing division ,we get $P = 1$ and $Q = 52^{\lceil \frac{L}{2} \rceil}$

For $0 \leq K \leq 26$:

In this case, each pair can be:

- $(uppercase, uppercase)$ for any character, i.e. total 26 pairs.
- $(lowercase, lowercase)$ for any character, i.e. total 26 pairs.
- $(uppercase, lowercase)$ for any of the K characters, i.e. total K pairs.
- $(lowercase, uppercase)$ for any of the K characters, i.e. total K pairs.

Considering all these possibilities, we have total $(26+26+K+K) = (52+2*K)$ possibilities for every single pair.

Thus, total number of palindromes will be:

- $(52+2*K)^{\lfloor \frac{L}{2} \rfloor}$ if L is even.
- $52*(52+2*K)^{\lfloor \frac{L}{2} \rfloor}$ if L is odd, since we can have any of the 52 characters at the center.

Hence, $numerator = (52 + 2 * K)^{\lfloor \frac{L}{2} \rfloor}$ and $denominator = 52^{L-(L \mod 2)}$, since 52 will be canceled out from numerator and denominator for odd L.

Since complete division is concerned, the result will be same even if P and Q are not co-primes. Thus, $P = numerator$ and $Q = denominator$.

Since $m = (10^9 + 7)$ is prime, we can use Fermats's little theorem to calculate $P.Q^{-1} \mod m$.
Hence, $Q^{-1} \mod m \equiv Q^{m-2} \mod m$.

Therefore, the required probability will be $(P * (Q^{m-2} \mod m)) \mod m$.

Time Complexity : $O(\log m)$ for each string.

PREREQUISITES:

Tree DP, Combinatorics and Observation.

PROBLEM:

Given a tree with N nodes, each node marked with a value $A[i]$ and a value x, Find out the number of ways to remove a subset of edges such that each of the new tree formed has xor-sum of all values in tree equal to x. Print this number of ways modulo $10^9 + 7$.

Let us call a subtree with xor-sum x as special.

QUICK EXPLANATION

- If the xor-sum of all nodes is not x or 0, it is not possible to achieve xor sum x by removing any subset of edges. The number of ways being zero in this case.
- Now, If a node has subtree xor-sum x, it shall be divided into an odd number of special trees, removing an even number of edges. Alternatively, if xor-sum of the subtree of a node is zero, it shall be divided into even number of special trees, by removing an odd number of edges.
- For each node, we can calculate the number of ways to remove odd as well as even number of edges such that each of the generated trees is special. For now, ignore the fact that subtree xor of current node may differ from 0 or x.
- Number of ways to remove the number of edges from the subtree of a node is an XOR-convolution of the number of ways to remove edges from the subtree of each of its children.
- If subtree xor of a node (not being the root) is x, the number of ways to remove an odd number of edges become the sum of the number of ways to remove odd and even number of edges. Similarly, if the subtree xor-sum is 0, the number of ways to remove even number of edges become the sum of the number of ways to remove odd and even number of edges.
- Corner case $x = 0$. Here, the number of ways is 2^x where x is the number of nodes with subtree xor being zero excluding root, only if subtree xor of the whole tree is zero. Otherwise, there is no way to split given tree into special subtrees.

EXPLANATION

First of all, Let's handle the base case, where $x = 0$. Here, Suppose we an edge connecting a subtree with xor-sum zero to its parent. The subtree xor is not affected at all. Hence, for every node (except root) which have subtree xor-sum zero, can be removed independently. If there are x such nodes, considering each combination, there are a total of 2^x ways to remove edges.

Coming to general case, we can notice that $x \oplus y \oplus y = x$. This means, that if from the subtree of a node, we disconnect even number of special subtrees, the xor-sum of the tree is not affected. But if an odd number of special subtrees is disconnected, the xor-sum of remaining nodes in the subtree is XORed by x.

Let's denote DPE_u denote the number of ways to remove even number of edges from subtree of node u **and edge connecting u and its parent** (except for root), while DPO_u denote the number of ways to remove odd number of edges from subtree of node u **plus the edge connecting node u and its parent**. It's clear that initially, $DPE_x = 1$ (We have the DP state a bit unusual, but it eases implementation. It's possible to solve it normal way too).

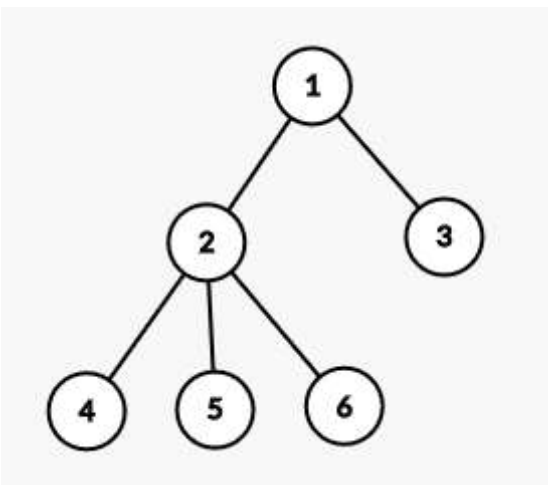
Suppose a node x has two direct children, say u and v. It can be easily seen, that $DPE_x = DPE_u * DPE_v + DPO_u * DPO_v$ while $DPO_x = DPE_u * DPO_v + DPO_u * DPE_v$. Can you spot the pattern? It's basically the xor convolution. We can also visualize it as the polynomial multiplication, where each node is represented by a variable of degree 1, x^i being the number of ways to remove i edges.

Now comes the trouble. We have the option of removing the edge connected current node with its parent (assuming current node is not root). If we do not remove that edge, the number of ways is just the XOR convolution of the number of ways to remove edges for children of the current node.

Suppose we have subtree xor of current node as x. In this case, if we were removing even number of edges, we always have the choice to remove an odd number of edges by removing the edge connecting current node with its parent, thus removing an odd number of nodes for each way we can remove an even number of nodes. Thus, we increase DPO_x by DPE_x if the subtree xor of a node is x and the node is not root.

Similarly, if the subtree xor of a node is 0, and we have removed an odd number of edges, we have the choice to remove the edge connecting the current node to its parent resulting in removing an even number of edges. Hence, if subtree xor of the node is zero, we can remove an even number of edges for every way we were able to remove odd number of edges. Hence, we can simply increase DPE_x by DPO_x if the subtree xor of the node is 0 and the node is not root.

Consider example tree where node 2, 3 and 6 are labeled 1, and rest are labeled zero. $x = 1$ for this example.



For node 6, $DPE_6 = 1$ as we may not remove edge 2-6. $DPO_6 = 1$ if we choose to remove edge 2-6.

For node 5, $DPE_5 = 1$ as we may not remove edge 2-5. $DPO_5 = 0$ as we cannot remove any odd number of edges, making special subtrees.

For node 4, $DPE_4 = 1$ as we may not remove edge 2-4. $DPO_4 = 0$ as we cannot remove any odd number of edges, making special subtrees.

For node 3, $DPE_6 = 1$ as we may not remove edge 1-3. $DPO_6 = 1$ if we choose to remove edge 1-3.

For node 2, we have to perform XOR convolution of number of ways of all its children, where DPE is coefficient of x^0 and DPO is coefficient of x^1. Clearly, we have $(1 + 1 * x^1) * (1 + 0 * x^1) * (1 + 0 * x^1)$ which equates to $(1 + x^1)$. This means, before considering edge 1-2, $DPE_2 = 1$ and $DPO_2 = 1$. Subtree xor of node 2 is 0, so, for every way to remove odd number of edges (excluding the edge connecting node and its parent), we can remove even number of edges if we choose to remove edge connecting node and its parent. So, we have $DPE_2 = DPE_2 + DPO_2 = 2$ and $DPO_2 = 1$.

For node 1, we once again need to perform the same convolution. We have the product as $(1 + 1 * x^1) * (2 + 1 * x^1) = (2 + 3 * x^1 + 1 * x^2)$ Important thing here to notice is that coefficient of x^2 represent number of ways to remove 2 edges. We only care about parity here, so polynomial $(3 + 3 * x^1)$ represent the same number of ways. Since this node is root, we cannot have any edge connecting this node to its parent.

Finally, xor-sum of the tree is 1, so, we can split this tree by removing an even number of edges, the answer to our problem is DPE_0, while if xor-sum is zero, the answer is DPO_1.

Those facing problem with this “convolution thing” refer the box below.

Click to view

You can simply think of this it as, parity of numbers after addition. Two odds make an even, two evens make an even, while one odd and other even makes an odd. I intentionally included this “convolution thing”. 😛

Time Complexity

Time complexity is $O(N)$.

PROBLEM

You are given a tree with N nodes numbered from 1 to N. A set S of nodes is called valid if there exist two vertices u and v (possibly, $u = v$) such that every node in S lies on the simple path from u to v.

Count the number of valid sets modulo $10^9 + 7$. Two sets are different if one node is included in one set and not in the other. If there are multiple pairs (u, v) making a set valid, that set is still only counted once.

QUICK EXPLANATION

- For each node, find the number of ways to select a subset of nodes in subtree of the node, such that all nodes can lie on a single path starting from that node. Let's call these paths as downward paths
- All paths in tree can be seen as combination of at most two such downward paths starting from a single node and going into different children subtrees.

EXPLANATION

Let's consider this problem subtask by subtask.

Subtask 1

Let us think of the shortest path covering a given subset. We can prove that such path would be unique for a chosen subset and shall have both of its endpoints inside the subset. If not, we can further remove nodes on endpoint which is not included.

Hence, if we denote value of a path (u, v) as 2^{L-2} where L is the number of nodes on simple path from u to v if $u \neq v$ and 1 if $u = v$, the answer to the original problem is the sum of values of paths over all unordered pairs (u, v).

Why?

Let's group subsets by the endpoint pair of shortest path containing all the nodes of subset. Let's assume pair (u, v) is the shortest path, and has total L nodes including u and v.

Now, both u and v must be in subset, but remaining $L - 2$ nodes may or may not be inside subset. Hence, there would be 2^{L-2} subsets having (u, v) as the shortest path containing the subset.

Note that this holds only when $u \neq v$. In case $u = v$, Only subset is $\{u\}$

Hence, for solving subtask 1, we can try all pairs of paths, compute distances in order to quickly compute values. Excluding the value of paths (u, u), each path might be included twice depending upon implementation.

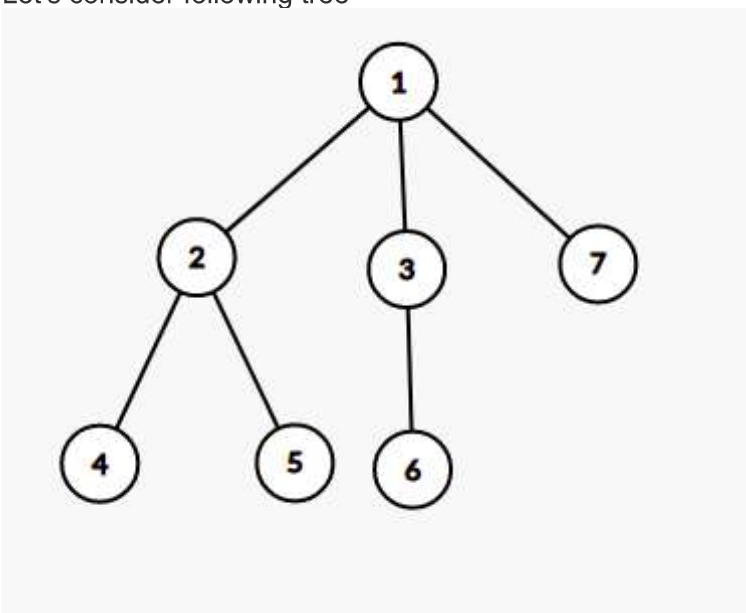
We can also try rooting the tree at node u and summing the value of paths from root to each node. This way, the depth of other endpoint would directly give the distance from root node.

Subtask 2

While the idea of summing the value of paths is good, we need to sum the value of paths in better than $O(N^2)$.

Now, let's root the tree at any node, and group the paths by the node having least depth among nodes lying on the path. We can see that for path (u, v), such node would be unique, and would be the Lowest Common Ancestor of u and v, say lca. For now, assume lca node is not u or v node.

Let's consider following tree



Let's count the sum of values of paths, which have node 1 as lca node. All paths can be of two types.

- lca node is one of the endpoint
- Two chains start from lca node and end in different child subtrees

Let's denote $f(u)$ as the sum of values of paths starting from node u and ending in subtree of node u, where node u may or may not be selected, but node v is selected. Let's refer to this as **modified value**.

Why defining new value: The benefit of using this definition is that we don't need to multiply value sums by 2 when u no longer remains endpoint (when some path is joined at node u)

For the paths having lca as one endpoint, It'll either be single node lca node path, or it will go into one of the child's subtree. Let's say immediate child ch is the child where the path ends. All paths starting from ch and ending in subtree of ch can be extended to include lca node.

For child $ch, f(ch)$ denotes the sum of values paths in subtree of ch, which can be extended to end at lca node. Hence, the number of such paths is $1 + \sum_{ch \in child(u)} f(ch)$. Also, It is easy to see that $f(u) = 1 + 2 * \sum_{ch \in child(u)} f(ch)$

Now, for paths not having lca node as endpoint, it shall be equivalent to selecting two children of lca node, and pair one chain in child one's subtree with one chain in second child's subtree. Say $ch1$ and $ch2$ are chosen. The sum of values of pairs of chains would be $f(ch1) * f(ch2)$. Hence, we need to compute $\sum_{ch1, ch2 \in child(u), ch1 < ch2} f(ch1) * f(ch2)$.

All these represent the subsets, but doesn't account for the lca node. lca node can either be included or not for each pair of chain. Hence, the sum of values of paths of second type becomes $2 * \sum_{ch1, ch2 \in child(u), ch1 < ch2} f(ch1) * f(ch2)$.

This quantity is easy to compute by considering children of u one by one and maintaing sum being sum of $f(ch)$ for all children of u already considered.

Hence, we have computed the sum of values of paths of both types, and also computed $f(u)$. Hence, we can solve this problme by a single DFS.

In case of doubts, please refer my implementation 1, which uses the same ideas line by line, and same variable names.

If interested, see an alternate implementation and try to prove how it works.

TIME COMPLEXITY

The time complexity is $O(N)$ per test case.

Prerequisites

Dp, Combinatorics

Problem Statement

You want to go attend your friend's party but the guard at the entrance stops you. He gives you a problem to solve and if you are able to do it, you get a free entry.
Given a sequence of n integers, you are allowed to perform the following actions: if p and q be two adjacent numbers, then you can remove them and write p + 2q instead of them. Here, p comes before q in sequence. You have to keep doing so until only 1 number is left. You have to get the biggest possible number.

The guard doesn't want to let you in. So he erases some of the numbers in a certain pattern.
He has m options, in the option j he erases all numbers to the left of the l_j-th number and all numbers to the right of r_j-th number. For each of the options he wants to know how big your final number is going to be.

This number can be very big, so output it modulo $10^9 + 7$.

Approach

Let the last number be negative. Then you want to minimize k_n. You can always can use $k_n = 1$. If that last number is non-negative, he can use $k_n = k_{n-1} + 1$. In this case, you can first merge the last two numbers. Thus, the sequence k_i consists of several blocks, in each of which $k_{i+1} = k_i + 1$, and each of the blocks except the first begins with 1.

Now we need to answer queries. We will do it offline. On the step i add the number $a_i$and and answer all queries with r=i. We will support the blocks for the query [1, i].

What happens when we add a number? If it's negative, it simply forms a separate block. Otherwise it becomes the end of some block. It is easy to see that the new block is the union of several old blocks.

How to answer queries? [l,r] is the union of some blocks and a suffix of another block. We can see that this is the partition into blocks for our query.

How to do it fast? We will store the boundaries of blocks to do binary search on them and find the block in which the *l* lies. We need to store the results for each block and prefix sums of these results. Also we need to find sums $\sum_{j=1}^{i} a_j 2^j$ to find out results for suffixes.

Each step is processed in O(log(n)) except for merging blocks. But we create at most one block on each step, so amortized time of each step is O(log(n)).

Also in this problem we need to be careful with overflows. Although we need to find the result modulo some prime, in some places the sign is important. We can use that if $|q|$ is more than maximum a_i, then p+2q is also big and have the same sign.

PREREQUISITES
Combinatorics,Maths

PROBLEM
There are n rose flowers and m tulip flowers in the garden . You have to form a flower bouquet using exactly t flowers, in which you have to use minimum of 5 roses and minimum of 2 tulip flowers. Find how many ways are there to form this flower bouquet.

EXPLANATION
Provided that we have to use minimum of 5 roses and 2 tulip flowers and in total t flowers,we can look for all possible combinations that can be made to form bouquet of t flowers using roses (5<=no of roses <=n) and tulips (2<=no of tulips <= m).
Example:-
if n=7 ,m=4, t=10
Since, we have to use minimum of 5 roses and 2 tulips and we have to use 10 flowers in this bouquet therefore we can select all 4 tulips along with 6 roses in 7 ways or the other combination would be taking all 7 roses along with 3 tulips in 4 ways . Hence the answer is (7+4) = 11 ways

PROBLEM:

You are given an array $A(|A_i| \leq 1)$ of size $N(1 \leq N \leq 10^5)$. For every x from $-N$ to N, you have to count the number of **non-empty** subsequences with a sum equal to x under modulo $163,577,857$

QUICK EXPLANATION:

- As we need to find out the sum of subsequences and we know that in this case, the order of elements does not matter. The only thing matter is the count of $-1,\ 0$, and 1.
- Let suppose, count of $-1,\ 0$, and 1 is $c_{-1},\ c_0$, and c_1 respectively. For every x from $-N$ to N, we can count the number of subsequences with a sum equal to x for $-1,\ 0$, and 1 separately and represent them as polynomials where power represents the sum of any subsequence and coefficient represents the count of subsequences with a particular sum. Now, the answer is the product of these polynomials.

EXPLANATION:

OBSERVATION:

- The $zero$ can not contribute to the sum of any subsequence and if there are c_0 $zeros$ in the given sequence then all the subsequences constructed using only these $zeros$ are given by 2^{c_0} and all of them have the sum equal to 0.
- As here the coefficients are going to be computed under modulo, so we use NTT over FFT and modulo mentioned in the question is also NTT friendly.

Let assume the count of 1 and -1 in the given sequence is c_1 and c_{-1} and define two polynomials:

First polynomial will be of the form $A(x) = a_0x^0 + a_1x^1 + a_2x^2 + ... + a_nx^{c_1}$, where a_i is the count of subsequences with a sum equal to i.

Similarly, the other polynomial will be of the form $B(x) = b_0x^0 + b_1x^{-1} + b_2x^{-2} + ... + b_nx^{-c_{-1}}$, where b_i is the count of subsequences with a sum equal to $-i$.

We know that the product of these two polynomials $P(x)$ is representing the sum of subsequences made by both 1 and -1. This polynomial will be of the form $P(x) = p_{-c_{-1}}x^{-c_{-1}} + p_{(-c_{-1}+1)}x^{(-c_{-1}+1)} + ... + p_{-1}x^{-1} + p_0x^0 + p_1x^1 + ... + p_{(c_1-1)}x^{(c_1-1)} + p_{c_1}x^{c_1}$, where p_i is the count of subsequences with a sum equal to i.

Now, we are going to compute the final answer, in which all the subsequences are considered. As the count of $zero$ is c_0 and subsequences with a sum equal to 0 and only formed using these $zeros$, are 2^{c_0}. We need to multiply this with each value of $P(x)$. To do that we can modify p_i like this:

$$p_i = 2^{c_0} * p_i$$

This includes the empty subarray too. we can remove it by just subtracting -1 from the p_0 and we can print the answer in the given formate.

ALTERNATIVE SOLUTION:

OBSERVATION:

- If we look closely then we find that polynomial $P(x)$ without the NTT. As there is a pattern, we can find a formula to compute each p_i for all valid i.

Let assume the count of $1, -1$ and their sum in the given sequence is equal to c_1, c_{-1}, and $c = c_1 + c_{-1}$ respectively.

We can interpolate that the value of p_i equivalent to choosing the choosing $(c_1 - i)$ items from the bag of c items. After we have $P(x)$, we can process the same as the above solution. See the tester's solution for further help.

COMPLEXITY:

TIME: $O(NlogN)$
SPACE: $O(NlogN)$

PROBLEM:

The *coolness* of a non-empty set of N distinct integers $S = \{x_1, x_2, \ldots, x_N\}$ is the count of integers X which divides $\prod_{i=1}^{N} x_i$ and primitive root [1] under modulo X exists.

Find the sum of coolness values over all possible non-empty sets having size in the range $[A, B]$, and elements in the range $[L, R]$.

QUICK EXPLANATION:

Consider the cases for values of X where primitive root under modulo X exists and separately derive formulas for each them for fixed size B.

For all subtasks, We can take summation over B , which becomes sums of $\binom{N}{R}$ values for fixed N. That can be calculated using square-root decomposition.

EXPLANATION:

Let's first see what exactly we have to calculate.

An integer X is nice if primitive root modulo 'X' exists.

So X must be one of the following :

$(1)\, X = 1$
$(2)\, X = 2$
$(3)\, X = 4$
$(4)\, X = p^k$
$(5)\, X = 2 * p^k$

where p is an odd prime and $k \in \mathrm{N}$.

(Source : Multiplicative group of integers modulo n - Wikipedia [3])

Now we can independently find the contribution of each of the 5 cases.

For now, consider a problem where we want to find total coolness of all subsets of size exactly B. and let's denote $K = \prod_{i=1}^{N} x_i$. K corresponds to product of elements of a set.

Case 1 : X = 1

1 is a divisor of every number, which means , for all the non-empty sets, 1 divides K.

Hence, this case contributes exactly 1 for every possible set.

$\therefore ans1 = \binom{R-L+1}{B}$

Case 2 : X = 2

2 divides K when set S contains at least one even integer. Conversely, 2 does not divide K when all the elements in S are odd.

So we consider all possible sets and subtract those having only odd elements.

If we denote, $Oddcount =$ Count of odd integers in the range $[L, R]$ = $R - R/2 - ((L-1) - (L-1)/2)$

then,
$ans2 = \binom{R-L+1}{B}$ - $\binom{Oddcount}{B}$

Case 3 : X = 4

This case contributes 1 to the answer if K is a multiple of 4.

$ans2$ already contains all the sets for which K is even, so we subtract the case when K is multiple of 2,but not 4.

For that, we choose $B - 1$ odd integers and one integer which is multiple of 2 , but not 4.

If, $Only2$ = Count of integers in the range $[L, R]$ which are divisible by 2 but not 4 = $R/2 - R/4 - ((L-1)/2 - (L-1)/4)$

then, $ans3 = ans2$ - $Only2$*($\binom{Oddcount}{B-1}$)

Case 4 : X = p^k

Consider only the odd primes. Then, each power of each odd prime from each integer will contribute 1 to the answer .

So for each integer in the range $[L, R]$, whenever it occurs in the set S, it's contribution will be sum of powers of odd primes in it.

Let's denote $Powersum$ = Sum of powers of odd primes of integers in the range $[L, R]$.

Here, we can calculate $Powersum$ for each integer using sieve and then take prefixsum.

$ans4 = Powersum$*($\binom{R-L}{B-1}$)

where, ($\binom{R-L}{B-1}$) is the number of times an integer in the range $[L, R]$ occurs in S.

Case 5 : X = 2*p^k

This is similar to Case 2, we add the answer we got in case 4 and subtract from it, the contribution of prime powers when K will be odd.

Let's denote
$Powersumodd$ = Sum of powers of odd primes of odd integers in the range $[L, R]$ = $Powersum(L, R) - Powersum(L/2, R/2)$

then, $ans5 = Powersum$* ($\binom{R-L}{B-1}$) - $Powersumodd$*($\binom{Oddcount-1}{B-1}$)

Now consider the problem when we have to find this answer for sizes 0 to B. We can solve that problem by replacing all the occurances of $C(N, R)$ by $S(N, R)$.

Where , $C(N, R)$ = ($\binom{N}{R}$) and $S(N, R)$ = $\sum_{i=0}^{R}$ ($\binom{N}{i}$)

So we have to calculate $S(N, R)$ values efficiently.
The main idea is to use square-root decomposition in some form. Below is one of the ways :

We know that ,

$C(N, R) = C(N-1, R) + C(N-1, R-1)$
$\therefore S(N, R) = S(N-1, R) + S(N-1, R-1)$

and $S(N, R) = S(N, R-1) + C(N, R)$ by definition

So we get, $S(N, R) = C(N-1, R) + 2 * S(N-1, R-1)$

This enables us to use square-root decomposition on N,

So we precalculate the $S(N, R)$ values when N is multiple of $BLOCK$, so we'll maintain $N/BLOCK$ rows.

Now for a query $S(N, R)$ we use the recurrance mentioned above, so N reduces by 1 each time and we'll get the answer in atmost $BLOCK$ steps.

Choosing $BLOCK$ as $O(\sqrt(N))$, the whole procedure works in $O(N * \sqrt(N))$ time and memory.

This was enough to pass within the time limit.

SOME OPTIMIZATIONS

(1) To solve one query for size atmost B, 5 different values of $S(N, R)$ are required, but it can be reduced to only 2 values and other 3 values can be calculated from these values in $O(1)$.

(2) This approach can also be performed offline by taking all the queries $S(N, R)$ and sorting them by N. Then we only have to maintain one row at a time, which will cost only $O(N)$ memory.

(3) If there are more queries, then it is better to use Mo's Algorithm here, which solves them in $O(MAXN\sqrt(MAXN) + Qlog(Q))$.

Outline of Mo's algorithm

The idea is that, the queries $S(N, R)$ will be represented as contiguous ranges, and we have to either change N or R by 1.

To change R, we just have to add or subtract some $C(N, R)$ value. To change N, we can use this recurrance $S(N, R) = C(N-1, R) + 2 * S(N-1, R-1)$.

SOLUTIONS:

Setter's Solution
Tester's Solution

VIDEO EDITORIAL:

Feel free to ask any doubts. Suggestions are welcome 🙂

1

Reply

Created	Last Reply	Replies	Views	Users	Likes	Links
Jan '21	Jan '21	2	1.5k	3	4	8

Quote Share

vruttant1403 Jan '21

Great Problem!!!

1

sevIII Jan '21

Idea about sqrt is genius! Thanks for amazing problem!

Reply

2

Share

Reply

Bookmark

Flag

Reply

You will be notified if someone mentions your @name or replies to you.

New & Unread Topics

Topic	Replies
DOTIFYPLAY - Editorial 2 editorial editorial greedy start92	3
BOX95 - Editorial editorial bitwise-operation editorial start95	7
UNRCOST - Editorial editorial binary-search editorial graphs greedy start97 strongly-connected-compone…	0
BLAST3 - Editorial editorial ad-hoc editorial start98	0
PRBW - Editorial editorial editorial segment-tree start125 trees	0

There are 37 unread and 6 new topics remaining, or browse other topics in editorial

PREREQUISITES :
Modular Exponentiation

PROBLEM :
Given M type of jewels we have to find the number of ways to place them in boxes with even number provided each box having number **i** will have **i** partitions.

EXPLANATION :
The number of empty slots are the summation of all even numbers upto N.
Let us assume x=floor(N/2) , then the number of empty slots are simply x*(x+1).
Each slot can be occupied by M different type of jewels thus the total number of ways are

WAYS = $M^{x*(x+1)}$

The above can calculated easily by using modular exponentiation while taking mod.

TIME COMPLEXITY :
The Time complexity is **O(log(N))**.

PREREQUISITES:

Modular Multiplicative Inverse 1 , Queries of nCr in O(1) time complexity 4 ,Fermat's little theorem 2

PROBLEM:

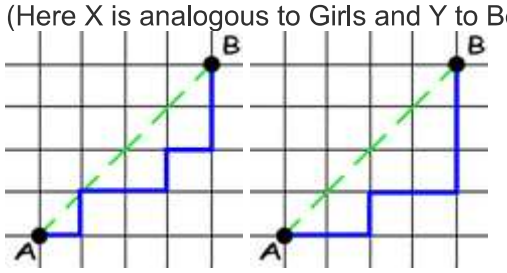
Find the number of ways in which in which each team of n and m members could arrange themselves , so that if G and B represents a string , then each prefix of string has more G than B. Then , we need to find the modulo of the "multiplication of both the number of ways" by $10^9 + 7$.

QUICK EXPLANATION:

This problem can be categorized as Dyck language 5 problem. So number of ways for achieving the required condition reduces to $2nCn$. So , then we just optimally need to find the multiplication of $2nCn$ and $2mCn$, and its modulo with $10^9 + 7$.

EXPLANATION:

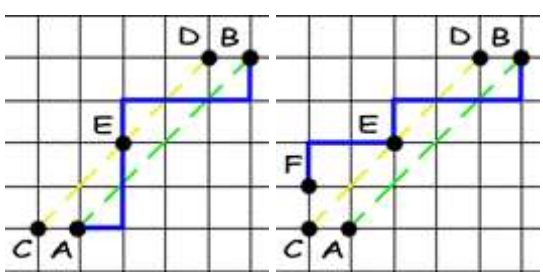
The goal is to find the number of Dyck words with a length of 2n. What is a Dyck word? It's a string consisting only of n X's and n Y's, and matching this criteria: each prefix of this string has more X's than Y's. For example, "XXYY" and "XYXY" are Dyck words, but "XYYX" and "YYXX" are not.
(Here X is analogous to Girls and Y to Boys)



Let's start the calculation process. We are going to build a geometrical analog of this problem, so let's consider paths that go from point A(0, 0) to point B(n, n) and do not cross segment AB, but can touch it (see examples for n=4).

We can just build a bijection in such a way: step right – 'X', step up – 'Y'.

Here's the main idea of the solution: Find the number of paths from A to B that cross segment AB, and call them "incorrect". If path is "incorrect" it has points on the segment CD, where C = (0, 1), D = (n-1, n). Let E be the point nearest to A that belongs to CD (and to the path). Let's symmetrize AE, part of our "incorrect" path with respect to the line CD. After this operation we'll get a path from F = (-1, 1) to B.



It should be easy to see that, for each path from F to B, we can build only one "incorrect" path from A to B, so we've got a bijection. Thus, the number of "incorrect" paths from A to B is $\binom{2n}{n-1}$. Now we can easily get the answer, by subtracting the number of "incorrect" paths from all paths:

$$\binom{2n}{n}-\binom{2n}{n-1}=\frac{(2n)!}{n!\cdot n!}-\frac{(2n)!}{(n-1)!\cdot(n+1)!}=\frac{\binom{2n}{n}}{n+1}$$

This number is also known as n's Catalan number.
So, now we know the number of ways for team A and team B are $2nCn$ and $2mCm$ respectively.
Now, we need to just multiply both the values of ways and find its modulo 10^9+7.
But as we know , the constraints are too big to follow this Naive approach,

$1 = T = 10^5$ & $1 = N, M = 10^5$

So we need an optimized way to calculate nCr % p, and one could notice the constraints, and conclude that an O(1) solution is required to get the solution. One could know the approach to do this by following this link 4 .
The idea behind is nCr can also be written as $(n!/(r! * (n-r)!))$ mod p , which is equivalent to $(n! * inverse(r!) * inverse((n-r)!))modp$. So, precomputing factorial of numbers from 1 to n will allow queries to be answered in O(log n).And, Precompute inverse of factorial in O(n) time and then quereies can be answered in O(1) time. Inverse of 1 to N natural number can be computed in O(n) time using Modular multiplicative inverse.
Using recursive definition of factorial, the following can be written as:
$n! = n * (n-1)!$
taking inverse on both sides
$inverse(n!) = inverse(n) * inverse((n-1)!)$

Below is the implemention of the above explanation

PREREQUISITES:

Modular Exponentiation [2] , Modular Multiplicative Inverse [4]

PROBLEM:

You have to find the number of ways of reaching $[n][m]$ in a $n * m$ grid from $[1][1]$ making exactly k turns turns along the way for each k, $1 <= k <= 2 * n - 2$.

EXPLANATION:

Subtask 1

We can use recursion to calculate number of ways.
We will start from 1,1 then we have 4 cases.

1. If we are moving in right direction then continue moving in it.
2. If
   1.
3. If
4. If
   1.

Quote Share

Code

```
#include<bits/stdc++.h>
using namespace std;
typedef long long           ll;

vector<int> turnAns;

ll n, m;

// 1 means right and 0 means down
// And prev means previous
void go(int x, int y, int currentTurns, bool prevTurn) {
  if (x == n && y == m) {
    turnAns[currentTurns] += 1;
    return;
  }
  if (x > n or y > m) return;

  // If prev turn is down
  if (prevTurn == 0) {
    // Move down
    go(x , y + 1, currentTurns, prevTurn);
```

Subtask 2

We can use Dynamic programming [3] aka DP to solve this problem.

We will make a 4 dimensional dp as dp[row][column][currentNoTurns][direction]

Then we will do transitions i.e. calculate values of our dp array as mentioned in the recursion part.

Code

```
#include<bits/stdc++.h>
using namespace std;
typedef long long           ll;

vector<int> turnAns;
```

```
ll n, m;
vector<vector<vector<vector<int>>>> dp;
const ll MOD = 1e9 + 7;

// Modular arithmetic operations
ll add(ll x, ll y) {ll ans = x + y; return (ans >= MOD ? ans - MOD : ans);}
ll mul(ll x, ll y) {ll ans = x * y; return (ans >= MOD ? ans % MOD : ans);}
ll sub(ll x, ll y) {ll ans = x - y; return (ans < 0 ? ans + MOD : ans);}

void solve() {
  ll ans = 0;
  cin >> n >> m;

  turnAns = vector<int>(2 * m - 1, 0);
```

Subtask 3

The First Observation to make is that once you reach the final row(nth row) or the final column(mth column), you don't have any choice but to make a turn towards $[n][m]$. Now the question reduces to finding the number of ways of reaching the final row or the final column in $k-1$ steps. A turn is defined as changing you trajectory by 90 degree towards the right(if you were moving down) or towards the bottom (if you were moving towards the right side). Therefore , you cannot make two top-down or left-right turns in succession
Since you can either start towards the ride side or the bottom one and have to reach either the nth row or the mth column , there will be 4 cases in total.

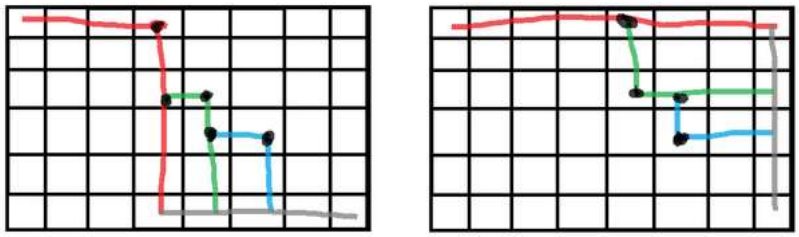
$Case\ 1$ (Starting towards the right and reaching the nth row):-

It is clear through observation that the minimum number of turns to do the same is 1 (A top down turn). Now ,since we can only make top-down and left-right turns alternatively, to reach the nth row , every time we make a left-right turn, we also have to make a top-down turn to shift the trajectory towards the bottom . Therefore , after the initial top-down turn we can only add turns in multiples of 2 to have a final top-down trajectory . Now since we initially made a top-down turn , the number of top down turns will always be one more than the left right turns. Let number of Top-Down Turns be R , then the number of Left-Right turns will be R-1. Now Since the total number of turns was $k-1$,

$R+R-1=k-1 => R=k/2.$

So, you have to make $k/2$ top down turns and $k/2-1$ left-right turns. Now the question reduces to finding the number of ways of choosing $k/2$ cells from a total of $m-2$ cells to make a top-down turn and choosing $k/2-1$ cells from a total of $n-2$ cells to make left right turn . This can easily be calculated by using the formula for combinations $(n-2Ck/2-1)*(m-2Ck/2)$.
It must also be noted that $k-1$ is an odd number since we started with 1 and added multiples of 2 and therefore $k\ must\ be\ even$ in Case 1.

$Case\ 2$ (Starting towards the right and reaching the mth column):-
It is clear through observation that the minimum number of turns to do the same is 0 and since you only add turns in multiples of 2 to maintain the initial trajectory , the value of $k-1$ must be even and the $value\ of\ k\ must\ be\ odd$. Since there will be an equal no. of top-down(say R) and left right turns(also R) ,the value of R will be $(k-1)/2$. Again , the question reduces to finding the number of ways of choosing $(k-1)/2$ cells from a total of $m-2$ cells to make a top-down turn and choosing $(k-1)/2$ cells from a total of $n-2$ cells to make left right turn . The answer for this case will therefore be $(n-2C(k-1)/2)*(m-2C(k-1)/2)$ and $k\ must\ be\ even.$



Case 1 Case 2

$Case\ 3$ (Starting towards the bottom and reaching the nth row):-
Through a similar process , it can be proven that the $value\ of\ k\ must\ be\ odd$ and the answer would be $(n - 2C(k-1)/2) * (m - 2C(k-1)/2)$.

$Case\ 4$ (Starting towards the bottom and reaching the mth column):-
The $value\ of\ k\ must\ be\ even$ and the answer would be $(n - 2Ck/2) * (m - 2Ck/2 - 1)$.

The final answer would be the sum of $Case\ 1\ and\ Case\ 4\ for\ even\ values\ of\ k$ and $Case\ 2\ and\ Case\ 3\ for\ odd\ values\ of\ k$.

You can easily calculate nCr for for smaller values in subtask 1 and for subtask 3, nCr for higher values can be calculated using Modular Exponentiation 2 and Modular Multiplicative Inverse 4 .

SOLUTIONS:

Setter's Solution
Tester's Solution
Editorialist's Solution

For doubts, please leave them in the comment section, I'll address them.

Reply

Share

Bookmark Flag

Reply

You will be notified if someone mentions your @name or replies to you.

New & Unread Topics

Topic	Replies
DOTIFYPLAY - Editorial 2 editorial editorial greedy start92	3
CS2023_404 - Editorial 6 editorial dynamic-programming editorial math start94	9
NUTRITION - Editorial editorial editorial start112	0
LEAFEAT - Editorials editorial	0
PREP48 Editorial editorial	0

There are 37 unread and 6 new topics remaining, or browse other topics in editorial

PROBLEM

All binary strings in this problem would be considered of length K, padded with zeros on left if needed. We start with an empty sequence of binary strings A, add binary representation of values in range $[L_i, R_i]$ at i-th query to A, and after each query, count the number of pairs (i,j) such that $1 \le i < j \le |A|$ such that the binary string $A_i \oplus A_j$ is a palindrome.

QUICK EXPLANATION

- By noticing that for two binary strings P and Q, $P \oplus Q$ being palindrome implies $P_i \oplus Q_i = P_{K-i-1} \oplus Q_{K-i-1} \implies P_i \oplus P_{K-i-1} = Q_i \oplus Q_{K-i-1}$ should hold for $0 \le i < K/2$ where P_i denote i-th character.
- So each binary string maps to a unique bitmask with $K/2$ bits, where i-th bit is generated by $P_i \oplus P_{K-i-1}$. All pairs of strings in the same equivalence class form a good pair, and no pair across equivalence groups form a good pair.
- Hence, If we can maintain the number of strings in each group after each update, and the number of good pairs before the update, we can easily compute the number of good pairs after the update.
- In order to avoid explicitly storing the frequency of each element individually, we can find groups of positions that shall always remain equal, compress them to speed up time, and save memory.

EXPLANATION

Notation:

- P_i denote i-th least significant character from string P, starting from 0
- $\oplus$ denote bitwise XOR operation.

Basic Intuition

Let's just consider two binary strings P and Q of length K. Assuming their XOR is a palindrome, we have for each i such that $0 \le i < K/2$, $P_i \oplus Q_i = P_{K-i-1} \oplus Q_{K-i+1}$ which implies $P_i \oplus P_{K-i-1} = Q_i \oplus Q_{K-i-1}$

Consider binary string P and build a mask M with exactly $K/2$ bits (floored down), where $M_i = P_i \oplus P_{K-i-1}$. Let's denote string M as the **mask** of string P. We can prove that each binary string of length K has a unique mask.

The XOR of two binary strings P and Q with K bits is a palindrome if and only if the mask formed by both strings is the same.

For some mask m, if X strings appear in A, it contributes $\binom{X}{2}$ pairs. As we can see that no pair is ever repeated across masks, the answer is simply $\sum_{m=0}^{2^{K/2}-1} \binom{f_m}{2}$ where f_m denotes the number of strings with mask m.

Naive Solution

This gives us an easy way to solve the problem. Let's keep a frequency array of masks and whenever we get an update, loop from L_i to R_i and update frequency and the total number of pairs. This solution can solve the second subtask, if used with a map, as storing a frequency array of size $2^{K/2}$ is not feasible.

Improving a bit

Let's assume for now that we can maintain the array in memory, but we need to perform updates quickly. An update of appending string $[L_i, R_i]$ can be seen as appending $[0, R_i]$ and then removing $[0, L_i - 1]$. So let's figure out appending $[0, R_i]$ first, the other case would be handled similarly.

Let's notice the string P and its generated mask carefully. Let's consider significant $K/2$ bits of P and reverse them, let's call it U (for upper). The bitwise XOR of U and lower $K/2$ bits of P is the mask of P. Consider interval of strings $[0, 2^{K/2} - 1]$, it contains all possible lower bits for fixed U. So the set of masks generated by XORing U with all strings is $[0, 2^{K/2} - 1]$. Read this line carefully.

The implication of last statement is that when appending all strings in an interval $[0, X]$, the frequency of all masks is increased by $\frac{X}{2^{K/2}}$, and for remaining $X - \lfloor \frac{X}{2^{K/2}} \rfloor * 2^{K/2} = X \bmod 2^{K/2}$ strings, the significant $(K+1)/2$ bits are fixed, while lower bits lie in range $[0, X \bmod 2^{K/2}]$.

Let's break down this $[0, X \bmod 2^{K/2}]$ interval into intervals of length power of two from left to right, greedily from largest power of two to smallest power of two. Now we need to find intervals in a trie manner, the continuous intervals of positions affected.

We can compute the actual intervals just like we use Trie to find the smallest XOR pair or finding a value in a list with k-th smallest XOR with the given value.

A walkthrough example of the above line

Once broken, let's find the actual intervals. Let's assume $K = 7$, consider $X = 62$, it's binary representation being 0111110, we can extract upper $K/2$ bits and reverse them to get $U = 110$. We need to increase frequency of all masks in $\{U \oplus x : x \in [0, 62]\}$. But considering all x in range $[0, \lfloor \frac{X}{2^{K/2}} \rfloor * 2^{K/2} - 1] = [0, 55]$, all masks appear $\lfloor \frac{X}{2^{K/2}} \rfloor = \frac{62}{8} = 7$ times.

Handling interval $[56, 62]$ now, all bits except last $K/2$ bits are same. Let's find $U = \text{reverse}(011) = 110$. Now since we only care for last $K/2$ bits of this range, it is equivalent to $[0, 6]$. We need to increase frequency of masks $0 \oplus U, 0 \oplus U \ldots 6 \oplus U$ by one. The masks are $[110, 111, 100, 101, 010, 011, 000] = [4, 5, 6, 7, 2, 3, 0]$ in order.

The interval $[0, 6]$ shall be broken into $[0, 3] + [4, 6]$ where $[4, 6]$ is further broken into $[0, 3], [4, 5], [6, 6]$. Each of these intervals would represent a continuous interval in our frequency array, whose start point depends upon U.

Now, considering bit b from $K/2 - 1$ to 0-th bit, if there's an interval with length 2^b, it shall have first b bits same as U and all subsequent intervals must have b-th bit opposite as U. Otherwise, all subsequent intervals must have b-th bit same as U.
In our example,

- There's an interval with length $2^2 = 4$, so It has 2-th bit, same as U,i.e. interval $[4, 7]$
- There's an interval with length $2^1 = 2$, so It has 1-th bit, same as U,i.e. interval $[2, 3]$
- There's an interval with length $2^0 = 1$, so It has 0-th bit, same as U,i.e. interval $[0, 0]$

As we can see, the above intervals represent the actual update needed.

Combinatorics

Now, We can update the frequencies of each mask efficiently. But Computing Number of pairs every time from scratch is still a pain. So let's notice that $\binom{x+y}{2} = \binom{x}{2} + \binom{y}{2} + x * y$. So if frequency of a mask m is increased by x,

then number of pairs is increased by $\binom{x}{2} + x * f_m$. Also, if frequency of a range $[L, R]$ of masks is increased by x, the number of pairs is increased by $\binom{x}{2} * (R - L + 1) + x * \sum_{m=L}^{R} f_m$.

Similar relations can be found for calculating the impact of removing one or X strings of same mask on pair count.

From our example, we had two kinds of updates to process on array $[0, 2^{K/2} - 1]$

- Increase all masks in a range $[L, R]$ by x
- In the above update, keep updating the number of pairs due to that update.

Segment Tree with Lazy propagation is a decent choice storing frequencies and a pair count variable.

Reducing memory Consumption

In all above, we assume we are able to store an array of length $2^{K/2}$, but Life ain't a bed of roses.

Meme

But We can notice that Each query can result in $O(K)$ updates, leading to a total of $O(N * K)$ updates. Hence, we can divide the frequency array into $O(N * K)$ disjoint intervals such that all frequencies in each interval remain the same, thus handling them simultaneously. A Little modification in Segment Tree to use the sum of weights of subtree instead of range length allows us to handle these weighted leaves. Refer to my implementation for details.

Important Note

This is one way to approach it. Users experienced with Tries may try dynamic Tries to avoid segment tree and coordinate compression, as all they need to maintain is the sum of frequencies.

TIME COMPLEXITY

The time complexity is $O(N * K * log(K))$ per test case with a high constant.
The memory complexity is $O(N * K)$ per test case since we only need to store updates $O(N * K)$ and segment tree $O(N * K)$ in compressed form.

Problem Link : ZIO 2013 Problem 4 5

Prerequisites : Graph representation, Permutation, and Combinations

Problem statement :

There are some tasks to be completed and some tasks are dependent on some other tasks i.e we have to do the dependent task before doing the current task (**Note :** it is not necessary to do it immediately before). Find the number of ways to complete all the tasks which satisfies the above condition

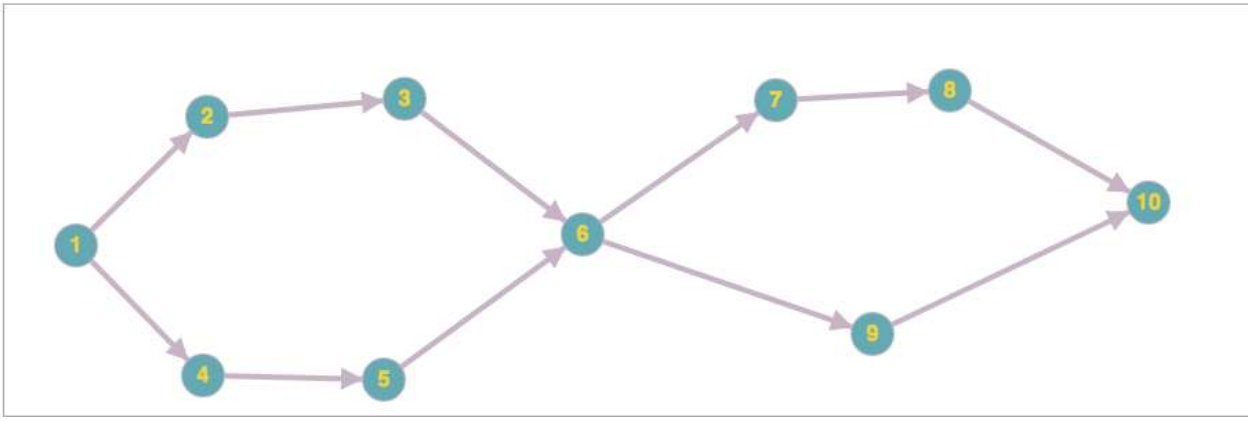
Idea :

We will represent this problem as a graph more precisely DAG (directed acyclic graphs) and group them and then find the number of permutations and compute the answer 🙂

Subtask a :

Task Number	1	2	3	4	5	6	7	8	9	10
Depends on	-	1	2	1	4	3,5	6	7	6	8,9

Representing this as graph :



Let us now do the groups here 🙂 →

In the above example , we see that 2 is dependent on 3 and 4 is dependent on 5 , but …… can we do 5 without doing 2 ? YES ! So we have to consider that combination also

Can we do 4 without doing 3? YES! We have to consider that also …

Can we do 6 without doing 3? NO

Can we do 6 without doing 5? NO

.

.

.

.

So as we can't do 6 without 2,3,4,5 … we don't include 6 in the group as we need to all of them before doing it

So what does a group consist of?

For any task in the group, there is at least one task which is not dependent on it. I hope you understood how groups are made… If you didn't understand then see the below examples for subtasks, it will be easy as there are graphs

and all 🙂

So making groups makes finding the number of these combinations easier? 🙂

YES

How?

Suppose we have 1,2,3,4: and

- 1 is dependent on 2
- 3 is dependent on 4

So now we will find the number of combinations of 1,2,3,4 satisfying the above combination :

1234

1324

1342

3412

3142

3124

So here are all the combinations we have to consider … So making groups help us find all the combinations

In the above example, we have tried all possible 4! I.e 24 test cases

But can we use permutation and combination and do it faster?

1,2 should be in order

3,4 should be in order

Note : Here 1 and 2 need not be consecutive they just have to be in order

So now, we have to choose 2 places for 1 and 2 and put 1 in the first place and 2 in the second place as 1 should be done before doing 2 and in this method, we will calculate all the possible positions for 1 and 2 and this can be done in 4C2 = 6

Now there are 2 places remaining out of 4 and we have 3,4 to fill it and as 3 should always be before 4, we should put 3 in the first place of the remaining 2 places and 4 in the second of the 2 remaining places and this can only be done in 1 way

Therefore, the total number of combinations is 4C2 = 6

So now you would have understood why this grouping works and how the number of combinations in a group is calculated …. So now let us move on and make the groups and find the answer 🙂

Now suppose we make

So here we make groups as follows :

1. Tasks 1 (we aren’t allowed to do any task without doing the first task)
2. Tasks 2,3,4,5 (we can do 5 without doing 2 and we can complete task 3 without doing task 5 but we can’t do any other tasks like task 6 without doing any of these 4 tasks , so these 4 are in one group i.e the permutation doesn’t matter for some of these 4)
3. Tasks 6 (we can’t do any other tasks without doing the 6th task so only 6 in one group)
4. Tasks 7,9,8 (To do the 10nth task we need to do 7,8,9 but we can do 9 task without doing 7th or 8th task , so these three are in one group)

5. Task 10 (To do task 10 , we need to do 8,9 , so task 10 is one group)

Now let us see the number of permutations we can do in each group

Note : we have to maintain the order i.e we can't do group 5 without doing group 2 so we can't permute the groups but we can permute the elements in the group)

Number of ways to do group 1 : 1 (1)

AND

Number of ways to do group 2 : 6 (4C2 = 6 combinations)

AND

Number of ways to do group 3 : 1 (6)

AND

Number of ways to do group 4 : 3 (3C1 = 3 combinations)

AND

Number of ways to do group 5 : 1 (10)

Total number of ways to complete all the tasks

Note : Here AND signifies that we have to do multiplication

1 * 6 * 1 * 3 * 1 = 18

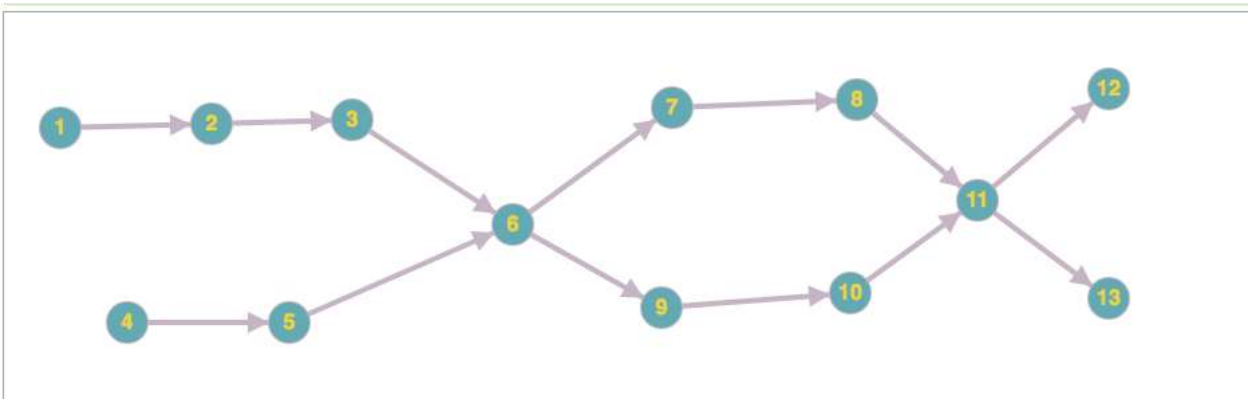
So we can complete all the tasks in 18 ways

Answer : 18

Subtask B :

Task Number	1	2	3	4	5	6	7	8	9	10	11	12	13
Depends on	-	1	2	-	4	3,5	6	7	6	9	8,10	11	11

Representing it as a graph :



Again as we did earlier, here we are going to make some groups as follows :

1. Tasks 1,2,3,4,5 (Tasks 1,4 are independent of each other and to do the 6th task we need to complete 3 and 5 but 5 is not dependent on 1,2,3 so we group these 5 tasks)
2. Tasks 6 (To do tasks 7,9 we need to have completed task 6 so we make this as a group)
3. Tasks 7,8,9,10 (Tasks 7,9 are dependent on 6 but tasks 9,10 are not dependent on tasks 7,8 and tasks 7,8 are not dependent on tasks 9,10 so we group these 5)

4. Tasks 11 (To do tasks 11 we need to have completed task 8,10 and both task 12,13 are dependent on task 11 , so we make 11 as a group)
5. Tasks 12,13 (Both of these are dependent on task 11 and neither of them are dependent on each other, so we group these two tasks)

Computing the number of ways to do the tasks :

Number of ways to do group 1 : 10 (5C3 = 10 combinations)

AND

Number of ways to do group 2 : 1 (6)

AND

Number of ways to do group 3 : 6 (4C2 = 6 combinations)

AND

Number of ways to do group 4 : 1 (11)

AND

Number of ways to do group 5 : 2 ((12)(13) , (13)(12))

Note : Here AND signifies that we have to do multiplication

Total number of ways to complete all the tasks

10*1*6*1*2 = 120

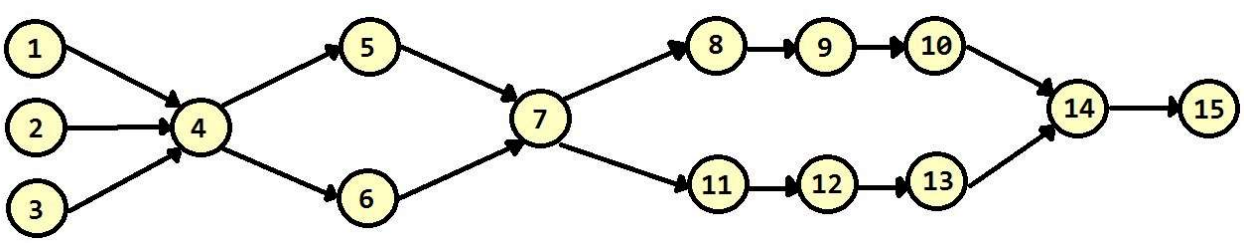
So we can complete all the tasks in 120 ways

Answer : 120

Subtask C :

Task Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Depends on	-	-	-	1,2,3	4	4	5,6	7	8	9	7	11	12	10,13	14

Representing this as graph :



So here we make groups as follows :

1. Tasks 1,2,3 (None of these three tasks are dependent on any of the other tasks therefore they are in one group)
2. Task 4 (there are no tasks which can be done without having the dependency of 4 so 4 is made a group)
3. Tasks 5,6 (we can do tasks 5,6 by 4 and 5 is not dependent on 6 and 6 is not dependent on 5 so 5,6 are grouped together)
4. Tasks 7 (task 7 is dependent on 5,6 and we can’t do any other task without doing so 7 is made one group)
5. Tasks 8,9,10,11,12,13 (tasks 9,10 depend on 8 and tasks 12,13 depend on 11 and 11,12,13 are not depended on 8,9,10 and tasks 8,9,10 are not dependent on tasks 11,12,13 so these 6 tasks are grouped together)
6. Task 14 (to do task 14 we have to complete tasks 10,13 and we cant do any task without 14 so 14 is one group)

7. Task 15 (to do task 15 we have to complete task 14 so 15 is in one group)

Now let us see the number of permutations we can do in each group :

Number of ways to do group 1 : 3! = 6 (123 , 132 , 231 , 213 , 312 , 321)

AND

Number of ways to do group 2 : 1 (4)

AND

Number of ways to do group 3 : 2 (56 , 65)

AND

Number of ways to do group 4 : 1 (7)

AND

Number of ways to do group 5 : 20 (6C3 = 20 combinations)

AND

Number of ways to do group 6 : 1 (14)

AND

Number of ways to do group 7: 1 (15)

Note : Here AND signifies that we have to do multiplication

Total number of ways to complete all the tasks

6*12201*1 = 240

So we can complete all the tasks in 240 ways

Answer : 240

Hope you understood 🙂

Special thanks to @avijit_agarwal for helping me do the (complex) maths and to @g_ajeet_11000 for helping me make the graphs so nice

PROBLEM

Given a string S of length N, count the number of subsets of set $S \subset \{1, 2, 3 \dots N\}$ such that there exists a string T similar to S having $T_x = c$ for all $x \in S$.

Two strings are considered similar if and only if both are of same length, and all corresponding substrings both are either palindrome, or non-palindrome.

QUICK EXPLANATION

- Merge pairs of nodes definitely same and add edges between pairs of nodes definitely not same, the resulting graph is a chordal graph, and we need something like number of independent subsets.
- Construct clique tree of above chordal graph, maintaining set of nodes of chordal graph represented by each node in clique tree.
- To count the number of independent subsets, we need to run dynamic programming on this tree, either choosing a specific chordal node of current clique, or not selecting at all.
- We also need to consider each chordal node representing a set of positions in original string.

EXPLANATION

This editorial has a lot to cover, so it may be hard to read, due to several corner cases.

The brute force approach is to enumerate all subsets, and checking each subset individually. Let's see how similar condition restricts us from choosing any string T.

Considering our string S, let's assume substring $S_{l,r}$ is a palindrome, but $S_{l-1,r+1}$ is not. This implies that in string T, character at position l and r should match, while characters at positions $l-1$ and $r+1$ **definitely** shouldn't match. This is all we need, find all non-equal pairs for S and for each subset, if it contains any pair, it is not a valid subset, getting 10 points.

Let's find all pairs which should be definitely equal (like $(l, r), (l+1, r-1)$ and so on, which appear as part of palindrome, and merge them into a single node.
Now find all pairs which should be definitely non-equal (like $(l-1, r+1)$) and add an edge between nodes representing position $l-1$ and $r+1$.

Now, the required number of subsets is as follows:

- consider all independent subsets of this graph.
- For each merged node present in independent subset, say x positions of string were merged at this node, so there are $2^x - 1$ ways to choose a subset, so we need to consider product of these over all nodes in current independent subset.

So generalizing, we need to solve problem similar to counting number of independent sets in this graph. For general graph, this is an unsolved problem, but our graph is special.

Lemma: The generated graph is Chordal Graphs [23] .

Proof

It is sufficient to prove that for every cycle of length greater than 3, there exists an edge not part of cycle, connecting two vertices of cycle.

Let's assume string $S_{l+1,r-1}$ is palindrome but $S_{l,r}$ is not. So we know, there's an edge between position l and r. Assuming $r - l - 1 > 2$, this implies positions $l+1$ and $r-1$ are definitely equal, thus merged into one node.

There are three cases

- Characters at position l and $l+1$ match: In this case, positions l and $l+1$ are merged, leaving only two nodes connected by a single edge.
- Characters at position r and $r-1$ match: In this case, positions r and $r-1$ are merged, leaving only two nodes connected by a single edge.
- Neither l match with $l+1$ position, not r with $r-1$. In this case, considering string $S_{l,l+1}$, there's edge $(l, l+1)$, similarly, there's edge $(r-1, r)$. But $l+1$ and $r-1$ are merged, say into node x, so we have edges (l, x), (x, r) and (l, r), leading to a cycle of length 3.

Hence, the generated graph is chordal.

Now, counting number of independent sets is done by building clique graph for given chordal graph, and running a dynamic programming.

The construction of clique graph is described in this 8 paper. I found This 4 and this 2 paper quite useful for understanding.

The gist of above papers (it's required to read first paper carefully, as it's page 10 contains algorithm to build clique tree). Som epointers which might be useful

- For each non-adjacent pair of nodes, the minimal separator is a clique.
- The maximal cliques over all separators represent the nodes of clique tree.
- For three cliques S_1, S_2 and S_3, such that edges connecting them are (S_1, S_2) and (S_2, S_3), there cannot exist any node of chordal graph, which was included in cliques S_1 and S_3, but not in S_2
- For two adjacent cliques S_1 and S_2, the intersection of the cliques is a minimal separator for all paths with one endpoint at each side.

I'd not cover the exact construction part, as it is explained in paper, and solutions attached uses same variable notations for ease of understanding.

Now, let's define some notation.
I'd refer nodes of chordal graph as chordal node and nodes of clique tree as clique nodes

- K: Number of clique nodes in clique tree T
- M: Number of nodes in chordal graph
- C_i: The set of chordal nodes present in i-th clique
- T_u: The set of clique nodes adjacent to clique node u.
- cnt_u: The number of positions of string merged into chordal node u

Now, we have several cliques, some adjacent cliques sharing vertices, and we need to compute number of independent sets of chordal graph.

Let's first deal with merged positions. Say x nodes are merged to form a chordal node. Assuming we find an independent subset of chordal graph, which contains this node. Including this node is equivalent to including any one of the non-empty subsets of x positions, thus giving $2^x - 1$ choices. These would be multiplied over each node present in independent set. Let us create $f(u)$ to automatically handle ways of including node u in independent set. So whenever we consider including node u into independent set, there are $f(u)$ ways.

It is important to notice that we can choose at most one node from each clique, but some cliques may share vertices, so that's why we are doing the following struggle

Let us try counting the number of independent subsets in subtree of clique node u, such that we select node x from current clique, denoting by $\text{chosen}(u, x)$, and number of independent subsets where we select no node from current clique by $\text{none}(u)$.

Considering adjacent cliques S_1 and S_2 such that S_1 is parent of S_2 (we can root the tree at any node), we can partition chordal nodes in $C_{S_1} \cup C_{S_2}$ into three disjoint classes:

- $C_{S_1} \cap C_{S_2}$: Appearing in both
- $C_{S_1} - C_{S_2}$: Appearing only in parent
- $C_{S_2} - C_{S_1}$: Appearing only in child

Assuming we select a node from $C_{S_1} \cap C_{S_2}$, we cannot select any other node from S_1 or S_2.

For computing $\text{none}(u)$, we need to consider all clique children v, and either we keep no node from clique node v, or some node from $C_v - C_u$, the third class in above classification. We can take sum of $\text{none}(v)$ and $\sum\limits_{x \in C_v - C_u} \text{chosen}(v, x)$, leading to $\text{none}(v) + \sum\limits_{x \in C_v - C_u} \text{chosen}(v, x)$ ways to chose independent subsets in subtree of child v, and we can take product over all children, getting following expression for $\text{none}(u)$

$$\text{none}(u) = \prod_{v \in T_u} [none(v) + \sum_{x \in C_v - C_u} \text{chosen}(v, x)]$$

Generalizing to multiple children of clique node u, consider all chordal nodes x in clique of clique node u, add x to P if no child clique contains node x, otherwise add it to Q

Mathematically, $P = C_u - \bigcup_{v \in T_u} C_u$ and $Q = C_u \cap [\bigcup_{v \in T_u} C_u]$

Now considering nodes in set P, we can effectively think of these as selecting no chordal nodes from clique u, and then select a node from P as it is independent from children cliques, giving us a simple way of computing $\text{chosen}(u, x) = \text{none}(u) \times f(x)$

Now, we need to deal with $x \in Q$, which may appear in one or more children cliques.

Let's consider each chordal node $x \in Q$ one by one and compute $\text{chosen}(u, x)$. For x, let's consider all clique children of node u. Two cases arise:

- If clique node v contains chordal node x, We must have counted ways to include it as $\text{chosen}(v, x)$, so we take it's product. But It also includes $f(x)$, so we consider $\dfrac{\text{chosen}(v, x)}{f(v)}$ ways.
- Otherwise, We either select no node from clique node v, or select some node of category 3, given by $none(v) + \sum\limits_{y \in C_v - C_u} \text{chosen}(v, y)$, already computed.

Lastly, we must multiply by $f(x)$ ways.

So we get recurrence $\text{chosen}(u, x) = f(x) * \prod\limits_{v \in T_u, x \in C_v} \dfrac{\text{chosen}(v, x)}{f(x)} * \prod\limits_{v \in T_u, x \notin C_v} [\text{none}(v) + \sum\limits_{y \in C_v - C_u} \text{chosen}(v, y)]$

This is all. The final answer would be given by $\text{none}(\text{root}) + \sum\limits_{x \in C_{\text{root}}} \text{chosen}(\text{root}, x)$. It is easier to add an empty clique node as root and attach it to one node, so answer would be $\text{none}(\text{root})$.

I hope this editorial, albeit most unreadable one I wrote, helps. Feel free to comment to ask doubts.

TIME COMPLEXITY

The number of edges in chordal graph cannot exceed $2 * N$ as we can think of it as fixing positions as center and extending palindromes till we get a non-equal pair, each centre contributing atmost 1 non-equal pair.

The computation for chosen function requires us to iterate over direct children of clique node and all chordal nodes of those children, leading to complexity to be of the order of $O(N^2)$ with/without a log factor, depending upon implementation.

PREREQUISITES:

Combinatorics, Binary Exponentiation 11 , Modular Inverse 21 .

PROBLEM:

You have to make a graph with N vertices and M edges. The graph follows the condition :

- For each vertex from i, where $2 \leq i \leq N$, A_i is defined such that the shortest path between node 1 and i is A_i and there is no other path with the same length as the shortest path.

EXPLANATION:

The first observation is that as there exists a path between node 1 and other nodes with length A_i so the graph is connected. So first let us consider $N-1$ edges and try to make all possible trees (connected graph with N nodes and $N-1$ edges is called a tree) following the condition specified in the problem.

Now note, if a node X has a path from node 1 of length A_X then it should be connected to a node with a node whose length from node 1 is $A_x - 1$. (`Using this condition you have to handle the test cases where there is no node which has the shortest path of length L and has nodes with length L+1 or L+2 or L+3 ... in that case the answer will be 0`). So we will add nodes in the increasing order of length of the shortest path i.e. first we will compute the ways to add nodes with the length of the shortest path 1, then 2 then 3 … soon. You can compute the number of ways in the following manner.

- There is only 1 node whose shortest path length is 0 which is the node 1 itself.
- We will initialize our answer variable with 1 as there is only 1 way to put node 1.
- Now suppose there are num_{L-1} nodes with shortest path length equal to $L-1$. Then the number of ways to add the node with shortest path length $= L$ will be num_{L-1} as I can connect this node with any one of the num_{L-1} nodes(`that's why you have to add nodes in increasing order of there shortest path length`) and we will multiply num_{L-1} with our answer variable.
- In this way you can compute the number way to add $N-1$ edges to the graph.

Now we will try to add remaining edges to the graph. You will note one thing that is the problem statement it is mentioned that shortest path between node 1 and node i is unique and all other paths are of greater length. If you rephrase this condition it means that you can only add the remaining edges only between nodes that have the same shortest path length.(`Why? because suppose if you add an edge between nodes with shortest path length L and L+1 then the node which already has the shortest path with length L+1 will now have 2 shortest path which violates the uniqueness condition and if you add an edge between nodes of shortest path length L and L+x where x > 1, then the node with the shortest path of length L+x will have a new shorter path of length L+1, again violating the condition`). Now to further compute the answer we will compute the number of ways to form a pair of nodes that are there such that both nodes of the pair has same length. This can be done by adding together $\binom{N}{2}$(number of ways to choose 2 objects out of N objects) for all the lengths of the shortest path.

```
long long node_pairs = 0;
for(int i = 1; i <= N-1; i ++) {
    long long count = shortest_path_length[i].size();
    node_pairs += ((count * (count-1)) / 2);
}
```

Firstly let us handle the corner test case where number of pairs is less than the number of extra edges i.e. $M - (N-1) > numOfPairs$ then the answer will be 0 as we can't add multiple edges between same pair. Otherwise, the number of ways to add these $M-(N-1)$ edges will be $\binom{numOfPairs}{M-(N-1)}$. To compute its modulo you will need the knowledge of modular inverse 21 . And we will multiply it with our answer variable to get the final answer. Now you might think that the magnitude of $numOfPairs$ can be quite large and to compute its factorial will result in TLE. so it can be computed efficiently as

- let quantity $M-(N-1)$ = X
- $\binom{numOfPairs}{M-(N-1)}$ = $\frac{numOfPairs!}{(numOfPairs-X)!*(X)!}$
- $\binom{numOfPairs}{M-(N-1)}$ = $\frac{numOfPairs!}{(numOfPairs-X)!}$ * $\frac{1}{X!}$
- $\frac{numOfPairs!}{(numOfPairs-X)!}$ = $((numOfPair - X + 1) * (numOfPair - X + 2) \cdots * (numOfPairs))$. It will have $M-(N-1)$ number of terms which can be computed in timelimit.(don't forget to do modulo when you are multiplying else will result in overflow).
- $\frac{1}{X!}$ % mod can be computed using modular inverse 21 .

TIME COMPLEXITY:

- Time complexity per test case is $O(N+M)$.

PREREQUISITES:

Dynamic Programming, Sqrt-decomposition

PROBLEM:

You are given n sets of integers $A_1, A_2, \ldots, A_n$. The sum of sizes of all sets is m.

Find the number of ways to choose numbers $a_1, a_2, \ldots, a_n$, such that $a_i \in A_i$, $a_i \neq a_{i+1}$ and $a_1 \neq a_n$.

QUICK EXPLANATION:

- The problem for a line can be solved with dynammic programming in $O(m)$.
- If $n \leq \sqrt{m}$, then count the number of ways by continously replacing pairs of adjacent sets with their intersection.
- If $n > \sqrt{m}$, then there is a pair of adjacent sets with intersection of size at most $\sqrt{m}$. Break the circle in the position of that pair of sets, and solve it as a line.

EXPLANATION:

First let's solve the problem when the sets are not in a circle, but in a line (so the first and last sets are not adjacent). Let $f_{i,x}$ be the number of ways of choosing elements from the first i sets, given that we are forced to choose x from the i-th set. We can calculate $f_{i,x}$ using the values from f_{i-1} as follows:

$$f_{i,x} = \sum_y f_{i-1,y} - f_{i-1,x}$$

From all the ways of choosing elements from the previous $i-1$ sets, we are removing the number of ways that chooses x as the element from set $i-1$ (because we can't have to adjacent elements with the same value). This dynamic programming runs in $O(m)$, it is implemented in the function `calc` in the tester solution.

Let L be the answer when the sets are in a line, i.e $L = \sum_x f_{N,x}$. Let's figure out how to extend it for the case when the sets are in a circle. L is an incorrect answer for a circle because it also counts some sequence of integers $x_1, x_2, ..., x_n$ ($x_i \in A_i$), where $x_1 = x_n$ (the first element equal to the last element).

Let I be the intersection of the first and last sets. To fix L we can just subtract all possible sequences of the form $y, x_2, x_3, ..., x_{n-1}, y$, where $y \in I$ i.e sequences with the same first and last element. We can count all such sequences by making $A_1 = A_N = \{y\}$ and running the previous dp.

The problem is that such algorithm runs in $O(|I| \cdot N)$, and $|I|$ can be very big. So the next natural question is to determine under which conditions is possible to find a bound for $|I|$.

It turns out that if the number of sets is greater than $\sqrt{m}$, then there exists a pair of adjacent sets with size at most $\sqrt{m}$, and we can solve such case in $O(m \cdot \sqrt{m})$.

The remaining case is when number of sets is at most $\sqrt{m}$. Let $C(A_1, A_2, ..., A_N)$ be the number of ways of choosing one integer from each set without taking the same element from adjacent sets (A_1 is adjacent with A_N). Similarly let $L(A_1, ..., A_N)$ be the number of ways when the sets are in a line (A_1 is not adjacent with A_N). Note that C stands for circle and L for line. The following recurrence holds:

$$C(A_1, ..., A_N) = L(A_1, ..., A_N) - C(A1 \cap A_2, A_3, ..., A_N)$$

Each step of the recurrence merges two adjacent sets to force the algorithm to choose an element from the intersection. In each iteration the number of sets decreases by one, therefore the algorithm runs N times, and since $N < \sqrt{m}$, the overall running time is again $O(m \cdot \sqrt{m})$.

PREREQUISITES

Combinatorics (Maybe Dynamic Programming)

PROBLEM

Given five integers L, R, A, B, K, find the number of arrays containing K integers such that all values lie in range $[L, R]$ and bitwise XOR of all elements is in range $[A, B]$

Do note that the order of elements matter in distinguishing one array from other.

QUICK EXPLANATION

- Reducing XOR sum in range $[A, B]$ into two subproblems where XOR sum must be up to B.
- Fixing most significant bit b where there exist an element in array such that it's significant bits up to b-th bit differ from both L and R.
- Considering all possible number of elements having bits up to b-th bit same bits L, R, or b-th bit different, we can compute the XOR sum of significant bits and the number of permutations of such groups.
- For less significant bits, it is easy to count the number of candidates for each element, depending on whether bits up to b-th bit match L, R or neither, and take required powers of number of candidates.

EXPLANATION

Since number of arrays with XOR in range $[A, B]$ is same as number of arrays with XOR strictly less than $B + 1$ less number of arrays with XOR strictly up to A, So we can directly focus on counting number of K-length arrays with elements range $[L, R]$ and XOR $< B$

Let's handle the arrays where all elements are either L or R separately. Assuming k occurrences of L, thus $K - k$ occurrences of R, if the xor of these values is $< B$, all permutation of k occurrences of L and $K - k$ occurrences of R contribute to arrays (There are $\binom{K}{k}$ of them).

So now, for remaining arrays, we are guaranteed that there exists at least one element in array (say X) such that $L < X < R$.

Let's fix the bit b where considering significant bits up to b-th bit of L, R and all values in array, there exists a value X in array such that significant bits up to b-th bit of X differs from both L and R. We will count all such arrays separately for each fixed bit and then take total. It is easy to prove that there is no double-counting of arrays here.

Considering $l = L/2^b$, $r = R/2^b$, $bSig = B/2^b$ and $l_2 = L \bmod 2^b$, $r_2 = R \bmod 2^b$ and $bRem = B \bmod 2^b$, l, r and $bSig$ holds significant b bits of L, R and B, and l_2, r_2 and $bRem$ holds remaining bits of L, R and B.

Since all elements in array had first $b - 1$ bits same as either L or R, the first b bits of any element in array can only be one among $\{l, l + 1, r - 1, r\}$ ($l + 1$ happens only when l has last bit off, similarly, $r - 1$ is allowed only when r has last bit on.)

Let's consider all possible tuples (a, b, c, d) such that

- $a + b + c + d = K$
- Exactly a values in array have significant bits up to b bits same as l
- Exactly b values in array have significant bits up to b bits same as $l + 1$ ($b = 0$ if $l \bmod 2 = 1$, all elements must have first b bits as l)

- Exactly c values in array have significant bits up to b bits same as $r-1$ ($c=0$ if $r \bmod 2 = 0$, all elements must have first b bits as r)
- Exactly d values in array have significant bits up to b bits same as r
- $b+c>0$ (If $b+c=0$, then there's no element in array having significant bits up to b bits different from both L and R)

For each tuple, we can easily find xor sum of bits up to b-th bit of elements in array (It only depends upon a, b, c and d.) If it's strictly greater than $bSig$, the xor sum would exceed B irrespective of remaining bits, so we can skip.

We can also find the number of permutations of 4 values l, $l+1$, $r-1$ and r as multinomial coefficient. For remaining bits, there are three options.

- Element have significant bits up to b-th bit same as l (group a)
  These elements can have remaining bits in range $[l_2, 2^b-1]$
- Element have significant bits up to b-th bit same as r (group d)
  These elements can have remaining bits in range $[0, r_2]$
- Remaining elements (group b and c)
  These elements can have remaining bits in range $[0, 2^b-1]$

We can now take powers of the number of candidates for each class and multiply (dependent events) to get the required number of elements.

One last thing to consider is when significant bits up to b-th bit of generated XOR sum for some tuple is exactly same as significant bits up to b-th bit of B. In that case, one of the element in group b or c can be set to make any required XOR sum for remaining b bits. There are $bRem$ candidates for XOR sum of last b bits, and one of the element is determined by XOR sum, we only need to count the ways to fix remaining values.

I have intentionally used variable naming in editorial. It is consistent with the solution attached, for easy reference.

It is also possible to kick out one K factor here by noticing that only $b+c$ and the parity of b and c matters in a tuple, so we can directly iterate over that and multiply ways by the number of (b, c) with given sum and parities, but it wasn't required to get accepted.

More efficient solutions are also possible. What's the fastest you can get? Can you beat this 8 ?

PS: There's one different (slower) approach with dynamic programming where you fix tuple $(b, lo, hi, tight)$ denoting the number of ways that upto b-th bit, exactly lo elements have bits same as L, exactly hi elements have bits same as R and the $tight$ denotes whether based on bits upto current bit, the XOR sum is equal to B or smaller.

TIME COMPLEXITY

For each bit, we need to iterate over tuple (a, b, c, d) with sum K, leading to K^3 tuples for each bit, leading to time complexity $O(K^3 * log(B))$

PREREQUISITES:

Z function, combinatorics

PROBLEM:

Given a string and a query k, output the number of ways we can choose k equal substrings from the string.

SHORT EXPLANATION

Let $cnt[i]$ be the number of different substrings that occur in the string exactly i times. Then the answer for a particular $k(\leq n)$ will be :
$\sum_{i=k}^{n} cnt[i] * \binom{i}{k}$ we can calculate cnt[i] using suffix array/Z function.

EXPLANATION:

Let $cnt[i]$ be the number of different substrings that occur in the string exactly i times.

For example, for the string “ababa”, the array will be:
$cnt[1] = 4 \rightarrow$ {“ababa”, “abab”, “baba”, “bab”} occur exactly once
$cnt[2] = 4 \rightarrow$ {“aba”, “ab”, “ba”, “b”} occur exactly twice
$cnt[3] = 1 \rightarrow$ {“a”} occur exactly thrice
$cnt[4] = 0$
$cnt[5] = 0$

Now suppose we want the answer for $k = 2$, i.e. number of ways to choose 2 equal strings. How do we calculate this using the cnt array?

$cnt[1]$ is of no use as we need 2 equal string.
$cnt[2] \rightarrow$ There are 4 different substrings that occur 2 times. So, we can add 4 to the answer.
$cnt[3] \rightarrow$ There is only 1 substring that occurs 3 times. But we need only 2 times, so we can choose any 2 of the 3. So, $\binom{3}{2} = 3$

Summing them, for $k = 2$ we get 7.

Now let us focus on calculating the cnt array.
To calculate this let us loop through all suffixes of S.

For the suffix $P = S[i..N]$, let us calculate the array $Z[i..N]$ where $Z[i]$ is the maximal equal substring starting from i that matches a prefix of $S[i..N]$.

For example, for the above string consider i = 1, the whole string is the suffix “ababa”.
Here the Z array would be
$Z[1] = 5 \rightarrow$ maximum prefix matching of “ababa” and “ababa”
$Z[2] = 0 \rightarrow$ maximum prefix matching of “ababa” and “baba”
$Z[3] = 3 \rightarrow$ maximum prefix matching of “ababa” and “aba”
$Z[4] = 0 \rightarrow$ maximum prefix matching of “ababa” and “ba”
$Z[5] = 1 \rightarrow$ maximum prefix matching of “ababa” and “a”

How is this array useful for calculating cnt?

From the above array we can conclude that we have 1 substring which can be choosen atleast 3 times. How? Because there are 3 entries in the above array that are greater than or equal to 1.
Similarly we can deduce for 2 (2 times), 3 (2 times), 4 (1 time) and 5(1 time).
So we increment cnt[3] once, cnt[2] twice and cnt[1] twice.

We do this for all suffix of S. The only thing left is to observe that cnt[i] now actually has how many different strings can be choosen *atleast* i times. We want it to be exactly i times. This is simple: we just subtract from $cnt[i]$ ($cnt[i+1]$ + $cnt[i+2]$ … + $cnt[N]$).

We can compute the Z function in $O(n)$ time. See here 111 for how to do that.

Time Complexity:

There are N suffixes and we take $O(N)$ time for each suffix so total time is:
$O(N^2).

PREREQUISITES:

Bitmask DP, Combinatorics

PROBLEM:

Given is a graph with N nodes which is almost complete, i.e., all but M edges are present. Count the number of permutations $P[1..N]$ such that if the permutation was considered to be the order in which nodes are visited in the graph then it is a valid traversal. In other words, there exists an edge from $P[1]$ to $P[2]$, from $P[2]$ to $P[3]$, ..., i.e., from $P[i]$ to $P[i+1]$ for $i = 1$ to $N-1$.

EXPLANATION:

Subtask 1
Simply iterate over all permutations of the nodes and count the valid ones by checking the presence of edges between $P[i]$ and $P[i+1]$. This approach takes $\mathrm{O}(N!)$ time and is fine for this substask.

Subtask 2, 3, 4
Most of the counting questions can be solved using DP. Let us see how we can think about this particular problem in terms of optimal sub-structure and overlapping sub-problems, i.e., the two necessary and sufficient requirements of a DP.

Let us think of the problem now in a way that we get to a DP. We can see that the number of missing edges are extremely few, i.e., just 7. In other words, the graph is an almost complete one. 7 missing edges means we just need to worry about the arrangement of at max 14 nodes. The other nodes can be shuffled in any manner since they don't affect the correctness of a permutation.

Let us call the set of nodes which have certain edges missing $faulty_nodes$. So it is the relative placement of the nodes of this set. Let x and y be two nodes in this set. If x and y HAVE an edge between them, i.e., edge (x, y) is not amongst the M missing edges, then we needn't worry about how x and y are placed relative to each other in a permutation. This is our only concern is that the nodes which share a missing edge should not be adjacent to each other in a permutation. Even if x and y appear adjacent to each other, that won't make the particular permutation invalid.

But what happens if x and y do not share an edge? We have to make sure they don't appear adjacent to each other. How do we ensure that? We basically need to make sure that there always exists at least one element z such that z has an with both x and y. In other words, z behave as a bridge.

This gives us a hint towards the solution, i.e., the DP. We know that there only exist at maximum 14 nodes in the $faulty_nodes$ set. This means that we can iterate over all possible relative placements of the faulty nodes. Not clear? Let us take an example. Let the given graph have 5 nodes n_1, n_2, n_3, n_4, n_5. Of these 5, let us assume that edges (n_2, n_3) and (n_3, n_4) are missing, i.e., n_2, n_3, n_4 are faulty nodes.

In this case, there are 6 (i.e., 3!) possible relative placements of the faulty nodes in a permutation. There are (n_4, n_2, n_3), (n_4, n_3, n_2), (n_2, n_4, n_3), (n_2, n_3, n_4), (n_3, n_4, n_2) and (n_3, n_2, n_4). Note that these are relative placements of the faulty nodes only. This means that the remaining nodes, i.e., n_1 and n_5 can be placed anywhere between these faulty nodes to form a permutation P of the graph. If then each adjacent pair $P[i]$, $P[i+1]$ have an edge between them then the permutation is a valid one. Now let us pick up any one of the 6 relative arrangements and try to add the two remaining nodes in it.

Let's take (n_2, n_4, n_3). Now let us see where all we can fit the n_1 and n_5 nodes so as to form valid permutations. We must place one between n_4 and n_3 since they can't be together. There must be a node between them to act as a bridge. So one valid placement of n_1 and n_5 can be $(n_1, n_2, n_4, n_5, n_3)$. Another one can be made by swapping n_1

and n_5. Further another one can be made one by moving around n_1 in $(n_1, n_2, n_4, n_5, n_3)$, something like $(n_2, n_4, n_5, n_1, n_3)$.

Now, we have a way to count the number of valid permutations that can be made. First, we take a relative arrangement of the faulty nodes. Then between each pair of faulty nodes that have a missing edge, we must place at least one normal node to act as bridge. This way, we will be able to count all the valid permutations.

One way to go about this is to cycle through all the arrangements of the faulty nodes. For a particular arrangement, we check which all pairs of adjacent faulty nodes need a bridge element between them. If a pair x, y needs one, we "tie" a normal node z to the front of y. Thus, we treat pair z, y as one element. Now, we arrange the "modified" nodes in the faulty nodes set (modified as in tied with normal nodes where required) in the $(N-$ number of normal nodes used as bridges$)$ possible positions. The number of ways to do this further multiplied by the factorial of number of normal nodes (this is because any of the normal nodes can be used and bridge and they can be shifted around) gives the number of valid permutations for the particular relative arrangement. But this method has a problem. There can be total of 14! possible arrangements of the faulty nodes. This is not practical. We have to reduce the number of cases to consider.

This is where DP comes into picture. Let's maintain a DP with three states: $DP[mask][first_node][number_tied]$. What do these states indicate? $mask$ tells us which of the faulty nodes have been arranged; $first_node$ is the first node of the arrangement and $number_tied$ tells the stores the number of normal nodes that have been used as bridges, i.e., tied to a faulty node. The state $mask$ goes from 1 to $2^{14} - 1$, $first_node$ has 14 possible values and $number_tied$ also has 14 possible values at max.

Thus, $DP[mask][first_node][number_tied]$ gives the number of arrangements of the faulty nodes whose bit is 1 in the number mask such that the first node of the arrangement is $first_node$ and the $number_tied$ is the number of normal nodes used as bridges.

We can now formulate the recurrence:

```
let k = faulty_nodes.size()
let ans = 0 //counter variable
let normal_nodes = N - k;

for mask = 1 to (2^k-1)
{
    for first_node = 0 to k-1
    {
        for number_tied = 0 to k-1
        {
            //appending a new node to the beginning of the arrangement that
            //is given by mask.
            for new_first = 0 to k-1
            {
                if(new_first bit in mask == 0)
                {
                    new_mask = bitwise_or(mask, 2^new_first)
                    missing_edge = 1 if edge (new_first, first_node) missing else 0

                    //adding to the count of sequences starting with new_first and
                    //containing faulty nodes as the ones in the new_mask
                    dp[new_mask][new_first][number_tied + missing_edge] += dp[mask][first_r

                    //note that if the edge (new_first, first_node) is missing,
                    //we will have to tie an element to first_node, i.e.,
                    //increasing number_tied by 1.
                }
```

```
}
```

The editorialist's program follows the editorial. Please see for implementation details. The (n choose k) mod m has been calculated using inverse factorials since m in this case is a sufficiently large prime.

OPTIMAL COMPLEXITY:

$\mathrm{O}(M^3 2^M)$ per test case.

Prerequisites:

Combinatorics, Modular Inverse

Problem:

Given a set of moves. Find the number of ways to reach from one point to other in a lattice.

Explanation:

This problem was an extension of the normal lattice path counting problem.
In the original problem, we had to find the number of ways to reach from **(0,0)** to **(x,y)**. So, consider we had **x** horizontal arrows and **y** vertical arrows. All the possible arrangements of the arrows was the answer, which is **(x+y)!/(x!y!)**.

We see that we have to simply count the number of permutations with repeated elements.

For this question, instead of all arrows having the same length, we have different lengths. So, we simply count the frequency of horizontal and vertical moves. Remember not to mix the frequency of horizontal and vertical moves, since they have different arrows.

So, simply the answer would be **(N+M)!/Product of(frequency of each move)!**

Remember to use a map for counting the frequency since the range lies upto **1e9**.

PREREQUISITES:

Combinatorics

PROBLEM:

Given three positive integers **N**, **L** and **R**, find the number of non-decreasing sequences of size at least 1 and at most **N**, such that each element of the sequence lies between **L** and **R**, both inclusive.

QUICK EXPLANATION:

Number of non-decreasing sequences of size exactly **K** will be Choose(**K** + **R** - **L**, **K**). Where Choose(**N**, **R**) is number of ways to choose **R** elements from **N** distinct elements.
Summation of Choose(**K** + **R** - **L**, **K**) for all **K** between 1 and N (inclusive) will be Choose(**N** + **R** - **L** + 1, **N**) - 1.
In the question it is given that the answer should be printed mod (10^6+3) which is a prime number.
Choose(**N**, **R**) mod **P** for **N** > **P** can be found by using Lucas Theorem.

EXPLANATION:

Let us assume that our non-decreasing sequence is of length **K**, i.e in other terms the sequence a[1], a[2], a[3] …… a[**K**] and a[**i**] <= a[**i**+1] and **L** <= a[**i**] <= **R** for each **i**.
Let us replace each a[**i**] of the sequence with a[**i**] – **L**. Let **P** = **R** – **L**. Now each a[**i**] satisfies the condition 0 <= a[**i**] <= **P**.

Subtask 1:

We can solve this by using DP. Make a dp[][] array where dp[i][j] stores number of non-decreasing sequences ending with j and are of length i.
If a[**i**] = **j**, and a[**i** - 1] = **k**, then **k** should be <= **j**. So dp[**i**][**j**] will be sum of dp[**i** – 1][**k**] such that **k** <= **j**.
Boundary case : dp[1][**j**] = 1 for all **j**.
Calculate these values for all 1 <= **i** <= MAX_N and 0 <= **j** <= MAX_P. For each test case output the sum of dp[**i**][**P**] for all **i**.
Complexity = O(**N** * **P**) Where **P** = **R** - **L**.

Subtasks 2 & 3:

Now let us consider that there are **K** red balls lying in a row. Now for each **i**, add a[**i**+1] – a[**i**] blue balls between **i**'th red ball and **i**+1'th red ball in the row. Add a[1] blue balls before 1st red ball and **P** – a[**K**] blue balls after **K**'th red ball in the row.

Few observations:

1. We can see that there will be exactly P blue balls in the row.
   Proof: Total number of blue balls = a[1] + a[2] – a[1] + a[3] – a[2] + ……a[**K**] – a[**K** – 1] + **P** – a[**K**]. All the terms of sequence a[] will be cancelled out leaving us with **P**.

2. Two different sequences produce different permutations of red and blue balls.
   Proof: Let us assume the two sequences are a[1], a[2]…a[**K**] and b[1], b[2]…b[K]. Let **i** be the smallest index where a[**i**] != b[**i**]. Simply assume a[0] = 0 and b[0] = 0. So there will be a[**i**] - a[**i** - 1] blue balls between **i**'th and **i**-1'th red balls in the first permutation and b[**i**] – b[**i**-1] blue balls between **i**'th and **i**-1'th red balls in the second permutation. Hence, the two permutations are different.

3. For each permutation of red and blue balls there exists a sequence which produces it.
   Proof: The sequence can be generated this way, each a[**i**] will be equal to (total number of blue balls to the left of it). In this sequence a[**i**] will never be greater than a[**i**+1] because there any blue ball to the towards left of **i**'th red ball in the row will also be to the left of **i**+1'th red ball in the row. So the sequence is non-decreasing. As the total number of blue balls is **P**, the highest element of the sequence can be **P**. Hence all the elements lie in the range 0 to **P**.

4. Observations 2 and 3 are enough to say that the mapping between the sequences and the permutation of balls is a Bijection. You can read about Bijective functions here 4 .

So total number of sequences will be equal to total number of permutations. The total number of permutations of **K** red balls and **P** blue balls in a row is a standard result which is equal to Choose(**K** + **P**, **K**).

This much solution is enough to solve the second subtask. Iterate through each of **K** between 1 and **N** and add Choose(**K** + **P**, **K**) to the answer.

Actually we can reduce this summation to single term to solve subtask 3. The summation is Choose(**P** + 1, 1) + Choose(**P** + 2, 2) ….+ Choose(**P** + **N**, **N**). Add and subtract Choose(**P** + 1, 0) to the summation, which shouldn't change the value of the sum.

Choose(**P**+1, 1) + Choose(**P**+1, 0) = Choose(**P** + 2, 1) (Since it is known that Choose(**N**, **R**) + Choose(**N**, **R** – 1) = Choose(**N** + 1, **R**)).
Now Choose(**P** + 2, 2) + Choose(**P** + 2, 1) = Choose(**P** + 3, 2).
Now Choose(**P** + 3, 3) + Choose(**P** + 3, 2) = Choose(**P** + 4, 3).
… So on till **N**'th term gives Choose(**P** + **N** + 1, **N**).

So the final answer is Choose(**P** + **N** + 1, **N**) – 1.

You can find the best ways to calculate Choose(**N**, **R**) mod **P** here 50 .

PREREQUISITES:

Observation, Elementary Combinatorics, Modulo properties

PROBLEM:

A game is played between two players, each with a pile of $N \leq 10^5$ cards. In each turn, both players draw a card from top of their piles. The player having higher card value wins this round, gets both cards and inserts those to the bottom of his pile. If the value of drawn cards is equal, both players insert their card at the bottom of their respective piles. For each transfer, the value of card decreases by 1 and when it becomes 0, it is discarded from the game. The player ending up without any card loses the game. If the game is a non-ending game, the player having higher sum of cards win. If even the sum of cards is equal, the match ends in draw.

Find the number of ways to distribute the cards between the players before the start of the game such that there should be a winner.

QUICK EXPLANATION:

1. Observe that the card with the maximum value will not be transferred in the whole game.
2. The player having maximum number of maximum-valued cards wins the game.
3. We can assign the cards which are not maximum to any of the players, it will not matter(2 ways).
4. Suppose the frequency of the maximum valued card is X. Observe that if X is even, there are $2^X - \binom{X}{2}$ ways of distributing and if X is odd, there are 2^X ways of distributing the X cards.
5. Calculate the final answer by combining the above two points.

EXPLANATION:

Observe the following facts:

1. The card with the maximum value will not be transferred in the whole game because there is no other card which is valued higher than this. So if we assign a maximum valued card to player A, it will belong to player A throughout the game.

2. All the non-maximum valued cards are transferable.

3. If the maximum valued card(s) belongs to only one player, after a finite number of rounds, he will be able to make the other player cardless.

4. If we assign the maximum valued cards to both players(some cards to Chef and some to Chefu), then all the non-maximum valued cards will be transferred between the players a finite number of times till their value become 0 and those are discarded from the game.

5. If we assign the maximum valued cards to both players, then at the end of the game(after a finite number of rounds), only the maximum valued cards will be left in the game and hence, the winner will be the player who has more maximum valued cards(since his sum of value of cards will be higher)

So, for the game to end in draw, the only condition is that we have to distribute the maximum valued cards equally between Chef and Chefu.

Case 1 (The maximum valued card has odd frequency):

In this case, there is no way by which we can distribute the maximum valued cards equally between the players and hence, the game will not end in a draw in any possible distribution. So our answer for this case is 2^N.

Case 2 (The maximum valued card has even frequency):

Suppose the frequency of maximum valued card is X. Now, we can give all the non-maximum valued $(N - X)$ cards to any of the players, it doesn't matter. (2 ways for one card and 2^{N-X} ways for all these cards).
Now out of the maximum valued X cards, we have to select at least more than half of these and at most all the cards, and assign the cards to either Chef or Chefu. The number of ways to do this can be expressed as:

$2 \cdot \sum_{i=X/2+1}^{X} \binom{X}{i}$

This can also be written as $2 \cdot ((2^X - \binom{X}{X/2})/2)$ which is equal to $(2^X - \binom{X}{X/2})$.

So the final answer for this case will be: $2^{N-X} \cdot (2^X - \binom{X}{X/2})$

View the content below in case you didn't understand the above statement.

Click to view

Note that factorials, their inverses and powers needs to be computed modulo $10^9 + 7$

Time Complexity:

The time complexity is $O(n)$ per test case.

PREREQUISITES:

Combinations, Counting, Modular Arithmetic, Modular Multiplicative Inverse 4 .

PROBLEM:

You are given a string S containing lowercase English alphabets. You are also given a list of N names (which are strings too, containing only lowercase English alphabets). For each of these names, you've to calculate the total number of sub-sequences of string S which are anagrams of that name. Since this value could be very large, calculate this value modulo 998244535. This value is called as the score of the name.

Finally, print the name having maximum score, if there are more than one such names, then print the one which appeared first in the list. If all names have 0 score, then print **"No Research This Month"** *(without quotes)*.

You've to do this for M testcases.

QUICK EXPLANATION

The score of each name is, $\prod^{S.count(c)} C_{name.count(c)}$ for each unique character c in name. Here, ${}^{n}C_{r}$ represents the total number of ways to select r objects out of available n objects.

EXPLANATION:

By definition, if a two strings are anagrams of each other, then the frequency of occurrences of each unique character in both of the string must be same. For example, if two string A and B are anagrams of each other, and the character c occurs exactly x times in A then, c must occurs exactly x times in B and vice-versa.

Let's denote the count of occurrences of character c in string A by $A.count(c)$. Now, we want to count the number of sub-sequences of S which satisfy the the above condition. Hence,

- We can first count and store the frequency of occurrences of each of the 26 alphabets in S.
- Then, for each $name$ in the list, we will store the count of occurrences of each alphabet.
- Now, for each character c, we need to select exactly $name.count(c)$ occurrences out of available $S.count(c)$ occurrences in S. Total number of ways to select these for character c is ${}^{S.count(c)}C_{name.count(c)}$.
- Hence, the final score will be the product of above expression for each $c \in$ lowercase alphabets.

Some points to mention:

- The calculations might lead to overflow, hence, apply modulo operation after each step.
- After applying modulo operation, direct division will give wrong results, hence to calculate $\frac{A}{B} \mod 998244353$, you will have to calculate $(A * B^{-1}) \mod 9982443535$. Here, B^{-1} is the modular multiplicative inverse of B under modulo 998244353.

Time Complexity : $O(|S| + N * (|name| + 26 * log_2(9982443535)))$ for each testcase.