PREREQUISITES:
Basic combinatorics
PROBLEM:
You're given string S of length N and character X. You have to count the number of substrings of S which contain at least one instance of X.
QUICK EXPLANATION:
Calculate substrings which don't contain X instead.
EXPLANATION:
How much substrings does S have? It has $N-K+1$ substrings of length K thus the total number (summing from substrings of length N to substrings of length 1) may be written as

$$1+2+\cdots+N=\frac{N(N+1)}{2}$$

Now let's calculate number of substrings which don't have X and subtract it from total number of substrings. Assume that $p_1<p_2<\cdots<p_k$ are positions such that $S_{p_k}=X$. For convenience we may add to them positions $p_0=0$ and $p_{k+1}=N+1$.

If substring doesn't have X in it then it means that both its ends are between consecutive positions p_i and p_{i+1}. How much such substrings is there between p_i and p_{i+1}? Well, it's exactly the number of substrings in the string $S_{p_i+1}S_{p_i+2}\ldots S_{p_{i+1}-1}$. Its length is $len_i=p_{i+1}-p_i-1$ and amount of its substrings is exactly $\frac{len_i(len_i+1)}{2}$.
Subtracting these values for all i from 0 to N we'll get the answer:

```
vector<int> p = {-1};
for(int i = 0; i < N; i++) {
    if(S[i] == X) {
        p.push_back(i);
    }
}
p.push_back(N);
int ans = N * (N + 1) / 2;
for(size_t i = 1; i < p.size(); i++) {
    int len = p[i] - p[i - 1] - 1;
    ans -= len * (len + 1) / 2;
}
cout << ans << endl;
```

PREREQUISITES:

Bits, Basic Combinatorics

PROBLEM:

You are given an array B which represents the bitwise-ors of the prefixes of A. Find the number of possible A.

QUICK EXPLANATION:

If there exists an i such that the set bits in B_i is not a subset of the set bits in B_{i+1}, then the answer is 0. Otherwise, let x be the total number of set bits in all elements of B except for the last element. The answer is 2^x.

EXPLANATION:

Notice that $B_{i+1} = B_i \bigvee A_{i+1}$ (we get a prefix of $i+1$ by adding element $i+1$ to the prefix of i). If some bit is set in B_i, then it must also be set in B_{i+1} because bits in x can't be unset by taking the bitwise-or of x and some other value y. So the first thing we check is that the bits which are set in B_i are a subset of the bits which are set in B_{i+1} for all valid i, and if this condition is not true for some i, the answer is 0.

Code for checking subset

x is a subset of y is equivalent to the condition (x&y)==x (this is a well-known bit trick).

Assume that $B_0 = 0$ (this makes sense since the empty prefix should have a bitwise-or value equal to the identity of the operation bitwise-or, which is 0).

For all valid i, we need to make sure that $B_{i+1} = B_i \bigvee A_{i+1}$. Let's split the bits into three types:

1. The bit is not set in both B_i and B_{i+1}.
2. The bit is not set in B_i but it is set in B_{i+1}.
3. The bit is set in both B_i and B_{i+1}.

Note that there isn't a fourth case, otherwise the set bits of B_i would not be a subset of the set bits of B_{i+1}.

For a bit of type 1, the bit in A_{i+1} can only be 0. If it is 1, the bit would be 1 in B_{i+1}.

For a bit of type 2, the bit in A_{i+1} can only be 1. If it is 0, the bit would be 0 in B_{i+1}.

For a bit of type 3, the bit in A_{i+1} can be anything.

Let x be the total number of bits of type 3 over all pairs (B_i, B_{i+1}). Since we have 2 choices for each bit, the answer will be 2^x. It remains to find x.

We can find x simply by iterating over each pair (B_i, B_{i+1}) and counting the bits of type 3. However, there is a more elegant solution.

A bit p will appear in B_i for all i in the range $[l, N]$. The total count of bits in this range is $N - l + 1$. However, only the bits in range $[l+1, N]$ will be of type 3, so we need to subtract 1 for each bit p. Summing up over each bit p, x is the total count of bits in all elements of B subtracted by the number of bits p which ever appear in the elements of B. If a bit appears in B, it must also appear in the last element of B, so the number of bits p which ever appear in

the elements of B is equivalent to the number of bits in the last element of B. Thus, x is the total number of bits in B except for the last element.

PREREQUISITES:

Dynamic Programming, Combinatorics

PROBLEM:

You are given N lines with slope a_i, intercept b_i and colour C_i. Every colour has a strength value of V_i associated with it. You are also given an erase with power equal to K.

A triangle is called good if it is constructed by the **same** colour lines and has a **positive** area. You have to **minimise** the count of such good triangles formed by given N lines. To do so you can use the eraser to erase any line you want but if you want to erase some line with colour C_i then the power of eraser decreases by V_i.

QUICK EXPLANATION:

Let's try to solve the problem step by step:

- First we have to pay attention to the constraints of K.
- If $K = 0$ then we know that we need to solve the problem for each colour independently.
- If $K > 0$ then can we still use the same approach and merge the answer to solve the problem. As possibilities are a lot so we should use dynamic programming to merge the answer.
- Two parallel lines can not participate in the formation of any triangle. So, for each colour, we should club the parallel lines and then it is just to remove some of them based on the value of K and find the answer.
- If there are X groups of parallel lines for some colour then the answer would be the sum of the product of the frequency of all unordered triplets from X.
- As V is unique for some colour then deleting any line for any particular does not affect the power of K but if there are X groups of parallel lines with the same colour then it is always optimal to delete the one from the group that has less number of lines. We'll explain the proof later in the editorial.
- Now, if we can precompute the answer for each K for each colour then merging the answer is only a simple knapsack problem.

EXPLANATION:

Observation: As only those triangles are going to be considered who are formed with the same colour of lines and all the lines of that colour have the same strength so we can easily solve the problem for each colour independently.

Lemma: If two lines are parallel then they never participate in the generation of any triangle.

Lemma: If there are two parallel lines X and Y with the same colour then the number of good triangles formed from both them is the same.

Lemma: If there are X groups each containing $f_1, f_2, ..., f_X$ lines such that $f_1 \leq f_2 \leq \ldots \leq f_X$ then we should delete from the minimum size group means first delete lines from f_1, then f_2, and so on…

Mathematical Problem: Given a set of non-negative integers $A_1, A_2, ...A_N$, you have to compute the sum of the product of all unordered triplets of this set after performing K operations. In one operation, select any positive integer from the set, and subtract **1** from its value. Find the minimum possible sum of the product of all **unordered triplets**.

Claim: For each operation, select the **minimum** integer from the set and apply the operation on this integer.

Proof: Let's assume $A_1 \leq A_2 \leq ... \leq A_N$ and any optimum solution for this is $B_1 \leq B_2 \leq ... \leq B_N$. It is guaranteed that $B_i \leq A_i$ for each valid $i (1 <= i <= N)$.

Now, suppose there exists some $i < j$ such that $0 < B_i$ and $B_j < A_j$ (if there doesn't exist any, it is already under our claim). Now, we can add **1** to B_j and subtract **1** from B_i, with that difference between previous sum and the current sum being positive which is more optimal and we still have a valid solution, since $0 \leq B_i \leq A_i$ for each valid $i(1 <= i <= N)$ which has better total value than the previous optimal solution.

Let's divide the optimal answer B in three-set:

- P_1 = sum of all numbers from B except B_i and B_j
- P_2 = sum of the product of two numbers from B except for B_i and B_j
- P_3 = sum of the product of three numbers from B except for B_i and B_j

Previous sum $= P_3 + P_2 * (B_i + B_j) + P_1 * B_i * B_j$

Current sum $= P_3 + P_2 * (B_i - 1 + B_j + 1) + P_1 * (B_i - 1) * (B_j + 1)$

Current sum $= P_3 + P_2 * (B_i + B_j) + P_1 * B_i * B_j + P_1 * (B_i - B_j - 1)$

Difference = Previous sum - Current sum

Difference $= P1 * (B_j - B_i + 1)$

As $B_i <= B_j$ which implies that Difference should be **positive** that again implies that it is a better solution than the optimal one which is a contradiction.

Now, let's use these to solve the problem. So, we can first partition the lines according to their colour and for each colour, we are going to partition them according to a_i value(we are creating parallel line groups as a_i is the slope) according to lemma **1** and **2**.

Now, for each colour, we have the groups of parallel lines which need to be sorted first according to **lemma 3** and with this lemma, we try to find the answer by removing the lines according to increasing order of the size of the group.

Then the problem is simply applying knapsack over the colours and keeping in my mind the value of K.

TIME COMPLEXITY:

TIME: $O(N * K)$
SPACE: $O(N)$

PREREQUISITES:

Combinatorics, Modular Arithmetic.

PROBLEM:

Given three integers N, S and M, print the number of valid strings of length N using an alphabet set of size S. A valid string is a string whose suffices of size greater than 1 are not a palindrome. Print this answer modulo M.

QUICK EXPLANATION

- Build the number of non-palindrome strings of length N from small to large.
- If the number of non-palindrome strings of length i is given by $cnt[i]$, then $cnt[i] = cnt[i-1] * S - cnt[\lceil i/2 \rceil]$. Just calculate this for all $1 \leq x \leq N$ modulo M.

EXPLANATION

Let us reverse the strings. It is easy to see that answer remains the same. Now, we have to compute the number of strings for which none of the prefixes of length greater than 1 is a palindrome.

Considering strings of length 1, we have a total of S different strings.

Considering strings of length 2, we can have $S * S$ different strings, out of which there are S strings having a prefix of length 2, which is a palindrome. So, we have $S * S - S$ valid strings.

Things become interesting now. See, For all these strings, suppose we append one more character, we get $S * (S * S - S)$ different strings of length 3. Let us try to find out how many from these are valid.

See, all these string of length 3 have a non-palindrome prefix up to length 2. So, the only way any string from these shall be non-valid is when the string is a palindrome of length 3. The number of palindrome strings of length x we can have in the set is same as the number of non-palindrome strings of length $\lceil x/2 \rceil$ which is $S * S - S$ for $x = 3$
So, Number of valid strings of length 3 is $S * (S * S - S) - (S * S - S)$.

Assuming we know the number of valid strings of length $x-1$ and length $\lceil x/2 \rceil$ we can find the number of valid strings of length x. This gives us a linear time solution, to sequentially compute valid strings of length x denoted by $cnt[x]$ using recurrence $cnt[x] = cnt[x-1] * S - cnt[\lceil x/2 \rceil]$. All computations to be done modulo M.

A common mistake is to subtract $S^{\lceil x/2 \rceil}$ when asked to count the number of palindrome strings of length x. But here, the strings in the set have the property that none of the string is having any prefix being a palindrome. So, It makes sense to count only those strings whose any other prefix is not a palindrome, which is the same as $cnt[\lceil x/2 \rceil]$.

For the people on lookout of a different solution, there also exist another solution using some inclusion-exculsion with backtracking which works for $N \leq 50$ only. You may find it for practice if you wish to. 😛

Time Complexity

Time complexity is $O(N)$ per test case.

PROBLEM:

You are provided a string s of length N. Suppose t be the string formed by concatenating the string s K times, i.e. $t = s + s + \cdots + s$ (K times). We want to find the number of occurrences of subsequence “ab” in it.

Finding number of subsequences “ab” in a given string s.

Finding number of subsequences “ab” in a given string s is same as finding number of pair of indices $(i, j), i < j$ such that s_i = ‘a’ and s_j = ‘b’. The brute force way of iterating over all such pairs of indices i, j and checking the conditions s_i = ‘a’ and s_j = ‘b’ would be $\mathrm{O}(|s|^2)$.

However, you can do better. In fact, you can find this in a single pass over the string in $\mathrm{O}(|s|)$ time. Consider an index j such that s_j = ‘b’, suppose we want to find number of i's such that $i < j$ and s_i = ‘a’. It will be same as number of occurrences of character ‘a’ till position j. We can maintain the count of a’s by iterating over the array from left to right. This way, we will be able to find the answer in single iteration over the string s in $\mathrm{O}(|s|)$ time. Pseudo code follows.

```
cnta = 0;
ans = 0;
for i = 1 to |s|:
    if s[i] == a:
        cnta++;
    if s[i] == b:
        ans += cnta;
```

Can we use this idea directly?

Now we construct the string t and find the number of subsequences “ab” in it in $\mathrm{O}(|t|)$ time, which will be $\mathrm{O}(N \cdot K)$. Constraints of the problem say that $N \cdot K$ could go up to 10^9. So, this won’t work in time.

Towards a counting based solution idea.

Let $t = s_1 + s_2 + s_3 + \cdots + s_K$, where $s_i = s$. We call s_i the i-th occurrence of string s.

Let us view the occurrences of subsequence “ab” in t as follows. One of the below two cases can happen.

- “ab” lies strictly inside some occurrence of string s in t, i.e. “ab” lies strictly inside some s_i. We can find the number of occurrences of “ab” in s. Let us denote it by C. The total number of occurrences “ab” in this case will be $C \cdot K$.
- In the other case, “a” lies inside some string s_i, whereas “b” lies in some other string s_j such that $i < j$. Finding the number of occurrences of “ab” in this case will be same as choosing the two strings s_i, s_j ($\binom{K}{2}$ ways), and multiplying it by the number of occurrences of “a” in s_i (denote by cnt_a) and the number of occurrences of “b” in s_j (denote by cnt_b), i.e. $\binom{K}{2} \cdot cnt_a \cdot cnt_b$. As $s_i = s$, cnt_a will be number of occurrences of “a” in s and cnt_b will be the number of occurrences of “b” in s.

We can find C, cnt_a, cnt_b in $\mathrm{O}(N)$ time. Thus, overall time complexity of this approach is $\mathrm{O}(N)$ which will easily pass in time for $N \leq 10^5$. Pseudo code follows.

```
cnta = 0
cntb = 0;
C = 0;
for i = 1 to N:
    if s[i] == 'a':
        cnta++;
    if s[i] == 'b':
        cntb++;
        C += cnta;
// First case, i.e. "ab" lies strictly inside some occurences of s, i.e. s_i in t
ans = C * K
// The other case, i.e. "a" lies inside some string s_i, where as "b" lies in other string s_
ans += K * (K - 1) / 2 * cnta * cntb ;
```

PREREQUISITES:

Combinatorics and fundamental counting principle 439

PROBLEM:

Given an array A of N positive integers, we define $f(S)$ as the mex of subsequence S. Find the sum of $f(S)$ over all subsequences of A.

QUICK EXPLANATION

- The mex of a subsequence of length N cannot exceed N. So we can replace all elements greater than N with N.
- Fixing the value of mex, let's try to count the number of subsequences with specific mex.
- For a given mex, we need each positive value smaller than mex to be present at least once, value mex shouldn't be present, and values greater than mex do not affect us.
- If f_x denote the frequency of value x, the number of non-empty subsets of these f_x elements is $2^{f_x} - 1$. We need to select one of the subsets for each x less than current mex, giving us $\prod_{x=1}^{mex-1} 2^{f_x} - 1$
- To consider elements greater than mex, we can take any subset of those. If there are y elements greater than mex, it contributes 2^y subsets for each choice of subsets of smaller values.

EXPLANATION

First thoughts tell us that mex of an array of length N cannot exceed $N + 1$. The proof is easy, just try to build an array with mex $N + 2$

Let's try fixing the value of mex, say M such that $0 \leq M \leq N$ is the current mex, and trying to count the number of subsequences with mex M. Say the number of subsequences is A, it contributes $M * A$ to the sum of mex of subsequences.

Now, for mex M, we need each value $1 \leq x < M$ to be present at least once. Let f_x denote the frequency of x in A.

For each x to be present once, we can visualize it as a set with f_x elements and we need to count the number of ways to select a **non-empty subset**, which we know, is $2^{f_x} - 1$

Since the choice of subset for each x is independent of other x, by fundamental counting principle, the number of ways of choosing values smaller than M such that each value is present at least once is $\prod_{x=1}^{M-1} 2^{f_i} - 1$

Now, we know that M cannot appear in subsequence. Suppose g_M denotes the number of elements greater than M, each choice of a subset of g_M elements is valid. We get 2^{g_M} choices.

Hence, the final answer can be written as $\sum_{M=1}^{N+1} M * 2^{g_M} * \prod_{x=1}^{M-1} 2^{f_i} - 1$

For implementation, sorting the elements would make things easier. Refer to my code for more details.

TIME COMPLEXITY

The time complexity is $O(N * log(N))$ per test case due to sorting as well as computing powers.

PREREQUISITES:

Suffix Arrays 4 or Suffix Tree 2 , LCP Array 7 , Combinatorics and Disjoint Set Union.

PROBLEM:

Given a string S and an integer K, for each L $1 \leq L \leq |S|$,

- K times, select a substring of S of length L
- Find the probability that all the chosen K strings are pairwise distinct.

All the computations are done modulo 998244353

QUICK EXPLANATION

- Build suffix array and LCP array of the given string, and
- selecting K distinct objects out of N objects (include duplicates) can be seen as the coefficient of x^K in $\prod(1 + a_i * x)$ where a_i denote the frequency of each distinct object.
- For length L, there are total $|S| - L + 1$ suffices, out of which, some might have LCP $\geq L$. Considering L in decreasing order. Only when $L \leq LCP_i$ for some i that suffix at index i and suffix at index $i + 1$ in suffix array shall have same first L characters.
- We maintain using DSU, the size of each distinct subset, and simultaneously maintain the first $1 + K$ coefficients of this product. Whenever we get $LCP_i = L$, we divide this by $(1 + a_i * x)$ and $(1 + b_i * x)$ and multiply by $(1 + (a_i + b_i) * x)$
- finally, we consider all the $K!$ orderings and divide by the total number of ways to select L length substrings to get final probabilities.

EXPLANATION

A simple problem
Let's consider a different problem. You have N buckets, each of which contains A_i balls. Find out the number of ways to select $K \leq N$ balls, such that at most one ball is selected from a bucket.

Writing in terms of polynomial, we can see that the required number of ways is given by the coefficient of x^K in $\prod_{i=1}^{N}(1 + A_i * x)$ (One way to interpret this is that either we select no ball from the current bucket (in 1 way) or select one ball (in A_i)$ ways.)

The following illustrates how the coefficients of the above polynomial behave

```
polynomial:                 x^0     x^1         x^2             x^3
(1+a*x):                    1       a           0               0
(1+a*x)*(1+b*x):            1       a+b         a*b             0
(1+a*x)*(1+b*x)*(1+c*x):    1       a+b+c       a*b+(a+b)*c     a*b*c
(1+a*x)*(1+c*x):            1       a+c         a*c             0
```

Suppose we have coefficients of $P(x)$ representing a polynomial, we can find coefficients of $P(x) * (1 + a * x)$ in $O(K)$. Similarly, If we have coefficients of $P(x)$ such that $(1 + a * x)$ divides $P(x)$, then we can obtain coefficients of $P(x)/(1 + a * x)$ in $O(K)$ time.

So, we have a special DS, which stores a polynomial (Initially just 1) and supports

- Multiply a polynomial by $(1 + a * x)$ in time $O(K)$
- Divide a polynomial by $(1 + a * x)$ assuming $(1 + a * x)|P(x)$ in time $O(K)$
- Return coefficient of x^K in time $O(1)$

Coming back to the original problem now.

The required probability for a given L can be written as the number of ways to select K strings of length L (in any order) $\times K!$ (considering all order of selection) divided by the total number of ways to select K strings (given by $(|S| - L + 1)^K$)

Hence, for a fixed L, if C_L denotes the number of ways to select K distinct substrings of length L irrespective of the order of selection, then the answer for length L is given as $\frac{C_L * K!}{(|S| - L + 1)^K}$ (in modular arithmetic). Our task now is to compute C_L for each length L.

Let's iterate over L in decreasing order. For length L, let's add the suffix of length L into our DS (equivalent to adding $(1 + x)$ into our DS). Also, for length L, it might be the case that two suffices to have the first L character the same.

For example, consider string "ababc", considering two suffices "ababc" and "abc". Till length > 2, the two suffices remain different, but when $L = 2$, the two suffices have the same first L characters.

This hints towards Suffix arrays and LCP arrays. So, let's build the suffix array and LCP array. Also, let's maintain the current group size for each group using a disjoint set Union.

Let's iterate over length L in decreasing order. For all pairs of adjacent suffices, if they have LCP $\geq L$, we need to merge them into same group. Suppose the first suffix has group size a and the second suffix has group size b.

At this point, it is required to remove $(1 + a * x)$ and $(1 + b * x)$ from our DS and add $(1 + (a + b) * x)$ into our DS.

This is all we do. We iterate over all length L in decreasing order, add $(1 + x)$ for suffix of current length, merge all groups having $LCP == L$, and query for the coefficient of x^K for each length, which is the required value of C_L.

Learning resources
Suffix Arrays and LCP array: here 7 and here 3
Disjoint Set Union 1

Problem to try
KPRB 15

After-thought
Can this problem be solved using suffix automation or suffix tree directly? Share your approaches.

TIME COMPLEXITY

The time complexity is $O(N * K + N * log(MOD))$ per test case.

PREREQUISITES:

Math, Combinatorics, Sorting

PROBLEM:

Find the number of paths from $(X_{initial}, Y_{initial})$ to (X_{final}, Y_{final}) such that it always passes through all the given S coordinates and avoids all the given K coordinates.

EXPLANATION:

Building some base to approach the solution

Sort all the S points according to x-coordinate first and then by y-coordinate.
Append $(X_{initial}, Y_{initial})$ to start the array and append (X_{final}, Y_{final}) to end of the array. Lets name the array Points.

Preliminary case of no paths

If all $Points$ cells cannot be traversed, in the present order, then answer will directly be -1. But, even if this condition is satisfied, the answer can be -1 even later on.
To check for this case - traverse the array and check the following condition :
The x and y coordinates should be non-decreasing respectively throughout the array. If not answer will -1, since we can't return back, due to restriction of only moving forward in both x and y

Solution for all other cases

Now start traversing the points in the above order. Let $ways[0]$ be no. of ways of reaching $Points[1]$ from $Points[0]$, $ways[1]$ be no. of ways of reaching $Points[2]$ from $Points[1]$. The final answer will be multiplication of all ways$_i$ for all i = 0 to $Points.size() - 1$.

So now we have reduced the problems to solving for sub-grids. So basically, we now have to find no. of paths in a grid such that x cells are blocked. We will show how to do it for 1 such sub-grid.
Consider s $(s.x, s.y)$ as the starting point of that sub-grid and e $(e.x, e.y)$ as the end point.

The blocked cell at (x, y) will only affect all cells, which have both coordinates greater than or equal to (x, y). We will now propose a method to calculate the no. of paths to reach end point e.

How to solve for a sub-grid?
Some basic info

No. of paths from $(x1, y1)$ to $(x2, y2)$ are $\frac{(x2-x1+y2-y1)!}{(x2-x1)!(y2-y1)!}$. How? Try to figure it out, try for eg, $(1, 1)$ to (N, M).

Approach

Find all the blocked cells which lie within our sub-grid. Sort the blocked cells according to x first, then by y.

Append the end point e to this set of points. Lets denote this set of points as w.

Maintain an array res, where $res[i]$ denotes no. of ways to reach point $w[i]$ from starting point s. Following **code snippet** will make it clear as to how we calculate the required answer, by subtracting contributions from the blocked cells. Sorting is essential as to subtract the contributions in correct order.

Code snippet

$calc(x1, y1, x2, y2)$ calculates the no. of paths from $(x1, y1)$ to $(x2, y2)$ if there were no blocked cells. How does calc work ? See *basic info* section.

```
for(int j = 0; j < w.size() - 1; j++) {
    for(int l = j + 1; l < w.size(); l++) {
        if(w[j].f <= w[l].f && w[j].s <= w[l].s) // Acc to above mentioned approach, w
        res[l] -= res[j] * calc(w[j].f, w[j].s, w[l].f, w[l].s)));
    }
}
```

In this manner we have calculated $way[i]$ which will be equal to res[w.size() - 1]. After the $way[i]$ has been calculated, all you need to do is multiply ans by $way[i]$, i.e ans *= $way[i]$.

Calculate the answer for each sub-grid and multiply them to get the final result.

Time Complexity

Time complexity - $O(K^2)$

Things to look out for

- Even after we have checked for the preliminary no-path case, its still possible that ans comes out to be 0. The output is going to be -1 for such a case.
- Since the output is modulo $1e9 + 7$, be careful of any overflows in between computations.

SOLUTIONS:

PRE-REQUISITES:

~~*DynamicProgramming*~~ *Combinatorics* 199 , *Mathematics*

PROBLEM:

Given an array X consisting of prefix and suffix values of some array A, find how many arrays A exist for a given array X. Report your answer modulo $10^9 + 7$.

Note- For entire editorial, please take note of below conventions to avoid confusions:

- $P_i = Prefix[i]$
- $S_i = Suffix[i]$
- Sum = sum of all array elements
- $1 - based$ indexing is followed unless mentioned otherwise.
- $Suffix[i] = S_i$ = Sum of last i elements in A, **which is exactly same as how suf is defined in the problem**. The conventional definition of $Suffix[i]$ = Sum of elements from $[i, N]$ is NOT used in the editorial!!

QUICK-EXPLANATION:

Key to
of comb

Let the
(using
valid A'

Quote Share

If valid A's exist, then we will pair each unpaired X_i with its corresponding X_{2n-i}. Now, take note of following-

- One of these (i.e. X_i or X_{2n-i}) will be P_j and other will be S_{n-j} for original array A. We essentially have 2 choices here.
- If I fix one of X_i or X_{2n-i} to be some prefix P_j, then the other element is forced to be S_{n-j}. We do not have a choice here!

With these 2 in mind, we first keep a frequency or count of all the pairs we have. Let $y_1, y_2, ..., y_m$ be the frequency of the pairs, with y_m being frequency of pairs with $X_i == X_{2n-i}$. We use following intuition to find the answer:

- X_i can be Prefix or Suffix! X_{2n-i} will take the leftover option. So we have 2 choices except for when $X_i == X_{2n-i}$. This contributes a factor of 2^{N-1-y_m} to the answer.
- All the X_i's can be assigned indices in $(N-1)!$ ways if they are all distinct. With this in mind, we use the formula to arrange N objects where $y_1, y_2, ..., y_m$ are duplicates. This contributes a factor of $\frac{(N-1)!}{y_1!*y_2!*...*y_m!)}$ where y_i is frequency of the pair.

$$Final\ Answer = \frac{(N-1)!}{y_1!*y_2!*...*y_m!} * 2^{N-1-y_m}$$

EXPLANATION:

We will deal with following in the Explanation Section:

- What I feel is the necessary mindset if you want to solve this (and similar) question(s) easily.
- Play with the X array.
- Discussing the Combinatorics and Deriving the Formula.

So, lets begin without delay 🙂

The necessary mindset

" Omg this Q is solved by only X people in my division! "
" This MUST be dp. I do not know dp. Bye bye!"

Ditch the first viewpoint entirely. You are defined by YOUR skills. You know, eons eons ago when I was solving a particular long in summer 62 what happened?

I did P_1, P_2, P_3. Got only partials in P_4, P_5, thanks to my huge knowledge gap in time complexity and sheer stupidity, and I then fully solved P_6! Thats when I felt that hey, I can do more. I know its very tempting to just

come to long, solve what you can and then quit. I understand when people criticise that 10 days for 11 problems is perhaps not the best rate of learning.

But give some time. What I see many people doing, they do not invest enough time at all. If they cannot think of the solution within 15 minute they quit. They sometimes do not even pick up pen and paper. Cmon! How are you going to do that derivation? Or see that observation which can be seen when trying some test cases?

My second quote deals with misjudging the topic. Be OPEN. Do NOT be narrow- being narrow will hurt you everywhere- in interviews, studies and jobs all alike. Be open. It is fine to feel that it is a dp problem, but be open to possibilities! Just because the modulo is $10^9 + 7$ does not mean its a dp problem, does it?

I will be honest, I do not know if this can be solved via dp or not myself. Perhaps there can be some advanced tricks. IDK. But what I see, is, that very simple, obvious observations are used in this question and it is still medium. I see that this question just utilizes the very fundamental basics of combinatorics and is medium. I see a lot of people panicking and being narrow. I see a lot of peple losing hope and giving up! I also see the same people not coming back to read the editorial. Lets change that. 🙂

Playing with the X array

I will first begin with how I feel a good coder can begin with the problem. We are given the array X which has prefix and suffix sums jumbled up. What to do?

OBSERVE! The first step to solve any problem is OBSERVATION. Can we think of any special observation here?

No, wait! I am wrong at one place. Observations are MADE special by our approach. We will observe anything and everything we can and decide which ones can be made special by our approach 🙂 .

The first thing coming to my mind is, that, X must have P_N and S_1 must be in the array. And they are nothing but sum of the entire array A ! Great, we have found one constraint, that Sum must be present in X and must be present twice!

Now, a lot of elements can be present twice or more, so this doesn't really help. Can we somehow expand or work on this observation?

The above observation on Sum doesn't directly help, but opens up the mind to think of another critical observation -

$P_i + S_{n-i}$ is bound to be equal to Sum. This means that it must be possible to pair elements (except P_N and S_1) of X such that sum of each pair same and equal to Sum !!

From here, we are almost done playing with array X. We insert two $0^{'}s$ into the X for convenience in implementation, as that allows us to pair P_N and S_1 with the $0^{'}s$ (why?). To pair the elements, we simply sort the array X and pair X_i and X_{2n-i}. It can be proved that only these 2 could be paired together. (How?)

We maintain the frequency of each pairs and proceed to the next section 🙂

Discussing Combinatorics and Deriving Formula

We will first tackle the factor of 2^{N-1-y_m} and then the factorial one.

We keep the count of frequency of each pair $y_1, y_2, ..., y_m$ where y_i is the frequency of $i^{'}th$ pair and y_m represents count of pair which has $X_i == X_{2n-i}$. Note that only 1 such type pair will exist where $X_i = X_{2n-i} = Sum/2$ (i.e. y_m is the only variable needed to account for pairs where both elements are same).

Factor of 2^(N-1-y_m)

For each pair (X_i, X_{2n-i}), X_i can either represent P_i or S_{n-i}. Once X_i is fixed, X_{2n-i} will take the remaining choice.

Now, note this - When we added two $0^{'}s$ to X, we made total pairs from N to $(N+1)$. Now, among all these $(N+1)$ pairs, our above rule does not apply to the pairs $(0, Sum)$ as the $Sum^{'}s$ position is fixed! Hence, the 2 pairs are not counted in there.

Next, we know that if $X_i == X_{2n-i}$ then it really doesn't matter which is prefix or which is suffix. Hence, these y_m pairs are also counted out.

Hence, we have $(N-1-y_m)$ such pairs for which we have 2 options , i.e. assign X_i to represent prefix, or assign it to represent suffix. Hence, total factor of 2^{N-1-y_m}

The factorial factor in Ans

For every pair (X_i, X_{2n-i}), if we fix the position of X_i, the position of X_{2n-i} gets fixed as well. In other words, say X_i gets assigned index j (prefix or suffix - doesn't really matter), then X_{2n-i} has to be assigned index $N-j$.

Hence, essentially for each pair, we have only 1 independent element whose position we can freely decide. (Bonus - Why is this valid? Why doesn't this lead to duplicates?)

After adding the two $0's$, we have $(N+1)$ pairs in X. Out of which, we do count the pairs $(0, S)$ because their position is fixed. This leads to a total of $(N-1)$ pairs.

Now, if frequency of each pair is known, we can simply use the standard formula of arranging $N-1$ objects when $y_1, y_2, ..., y_m$ are same, which is $\frac{(N-1)!}{y_1!*y_2!*...*y_m!)}$. Note that if the pair is occurring only once, then its $y_i = 1$ so it has no effect on the formula.

Based on above sections, we get the final answer as-

$$Final\ Answer = \frac{(N-1)!}{y_1!*y_2!*...*y_m!} * 2^{N-1-y_m}$$

SOLUTION

Setter
Tester

$Time\ Complexity = O(NLogN)$
$Space\ Complexity = O(N)$

CHEF VIJJU'S CORNER 😃

Why 2 zeroes are inserted into X?
Why only Xi and X(2n-i) can be paired
Why can we chose position of Xi freely without worrying about duplicates
Setter's Notes
Kudos to Setter
????????
Related Problems

56

Reply

Created	Last Reply	Replies	Views	Users	Likes	Links
Jan '20	Jun '20	67	7.5k	40	102	26

Frequent Posters

Popular Links

199 Community - Competitive Programming - Competitive Programming Tutorials - Basics of C... topcoder.com
123 Contest Page | CodeChef codechef.com
89 Contest Page | CodeChef codechef.com
84 Contest Page | CodeChef codechef.com
62 Ranklist - MAY17 | CodeChef codechef.com

There are **67** replies with an estimated read time of **14 minutes.**

Show top replies

jha Jan '20

Can someone also explain the modular arithmetic in this question. I did exact same thing but only 20 points, rest were showing SIGFPE error, and that is due to division or mod by 0 or integer overflow.
Here's my solution

https://www.codechef.com/viewsolution/28960826 15

muskan1234 Jan '20

Can someone explain why am i getting TLE for similar approach.
https://www.codechef.com/viewsolution/28987176 30

Reply

raisinten jha Jan '20

Yuuup, the SIGFPE is being caused by a division by `0` `@line 61` of your solution.

You may go through my submission for CHEFPSA here 49 and at the top you'll find a link to a cute article explaining the precomputation part of the modular arithmetic involved in this problem.

Reply

I hope it becomes clearer when you see how I incorporated that in my code. 🙂

1

mahendra3844 Jan '20

Can any one help me my code passing only some of the test cases
I used two pointers approach to find pairs similar to the editorial
https://www.codechef.com/viewsolution/28988619 11

Reply

abhaypatil2000 satwik_bhv1 Jan '20

check "modular division blogarithms" you will get the proof of fermats theorem also

Reply

balrams jha Jan '20

GeeksforGeeks – 29 Apr 17

Modulo 10^9+7 (1000000007) - GeeksforGeeks 5

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview Questions.

smile_forever Jan '20

BEST EDITORIAL ever I seen 😍

Reply

2

rishabh1252 Jan '20

Thanks a lot for the great editorial !

Reply

1

sourabh_27 Jan '20

Best editorial till date I have read!! Thanks

Reply

1

i_64 Jan '20

I'm trying to submit my code in PRACTICE for CHEFPSA and I'm getting:
"Status: Error Message: No such contest exists, recheck the contest code and try again"
anyone else getting the same error??

Reply

Reply

PREREQUISITES:

Combinatorics 3 , Bit Manipulation (optional) 1

PROBLEM:

Arranging N numbers into M arrays (no element repeats in a single array) of equal size such that union of any K arrays result in the sample set N, but union of any number of arrays less than K doesn't satisfy this.

QUICK EXPLANATION:

Let us now consider this simple case, let us say we have chosen k−1 arrays from M arrays at random, and we were to choose 1 more array to get the complete set. We know that the k-1 arrays cannot complete the set by themselves and hence the remaining M-k+1 arrays must have an element that the k−1 arrays don't.

COMPLETE EXPLANATION:

You may have got the gist of the problem from the quick explanation above.
So the overall idea is to distribute the elements in M arrays such that M-k+1 arrays must have a number which the remaining k-1 arrays don't have.

So, here we will have C(M, k-1)combinations of each k-1 array groups. Hence N (no. of distinct elements) must also be C(M, k-1), any N other than this can result in no solution (But only the valid cases are considered in the problem). Therefore for a valid distribution, there should be M-k+1 copies of each element.

Distribute the elements by adding an element in M-k+1 arrays and exclude the element in remaining k-1 arrays.
To achieve this, you can consider C(M, M-k+1), and add an element to each selected array combination, preferably the current lowest element so that the answer itself is lexicographically non-decreasing.

To do the above in python is quite easy due to itertools module but I strongly recommend to try it on your own from scratch.

PREREQUISITES:

Modulo operator and Basic Combinatorics.

PROBLEM:

Given two integers N and P, suppose the maximum value of $(((N \bmod i) \bmod j) \bmod k) \bmod N$ be M where $i,j,k \in [1,P]$, Find the number of ways to select $i,j,k \in [1,P]$ such that $(((N \bmod i) \bmod j) \bmod k) \bmod N$ equals M.

SUPER QUICK EXPLANATION

- The maximum value of $N \bmod x$ where $x \in [1,N]$, if N is odd, is $(N-1)/2$ when $x = (N+1)/2$, and if N is even, is $N/2-1$ when $x = N/2+1$.
- We can achieve $(((N \bmod i) \bmod j) \bmod k) \bmod N = M$ in three ways. Let $x = \lceil (N+1)/2 \rceil$
  - $i = x$ and $j,k > M$.
  - $i > N, j = x$ and $k > M$.
  - $i,j > N$ and $k = x$.
    Each of this case can be easily computed.

EXPLANATION

First of all, Let is find this value M. It has to be less than $min(i,j,k,N)$ which implies, $M < N$. Hence, if we want $M > 0$, we need $(((N \bmod i) \bmod j) \bmod k) < N$. So, We know for sure, that to maximize M, $min(i,j,k) \leq N$. Hence, we need maximum $(((N \bmod i) \bmod j) \bmod k) < N$ and now we can ignore the last $\bmod N$.

So, The maximum $N \bmod x$ can attain is $\lfloor (N-1)/2 \rfloor$. This happens when $x = \lceil (N+1)/2 \rceil$. It can be easily verified either by checking by hand, or writing a simple program 😉

Now, try finding out number of ways $(((N \bmod i) \bmod j) \bmod k)$ equals M. It can be approached in Simple case base analysis.

We can try all possible triplets of (i,j,k) and generalize them into three cases.

- When $i = \lceil (N+1)/2 \rceil$ and $j,k > M$
- When $i > N, j = \lceil (N+1)/2 \rceil$ and $k > M$
- When $i,j > N$ and $k = \lceil (N+1)/2 \rceil$

In all three cases, we can simply count the number of triplets (i,j,k) satisfying any condition and print the answer.

Corner Case

When $N \leq 2$, $M = \lfloor (N-1)/2 \rfloor = 0$. This is because we cannot achieve $(((N \bmod i) \bmod j) \bmod k) \bmod N > 0$. So, all triplets (i,j,k) are valid.

Alternate solution - read at your own risk, you have been warned 😃

For those curious enough not to be satisfied with such solutions, there also exists a pattern based solution too, using basic math. Just use brute solution to find the first terms of series and solve using the pattern formed. Number 6 is important. Enjoy 😃

Time Complexity

Time complexity is $O(1)$ per test case.

PREREQUISITES:

Dynamic Programming, Combinatorics, Stirling Numbers

PROBLEM:

Given C identical charities and N unique coins, find the number of ways of distributing the N coins to the C charities such that each charity gets at least 1, 2 or 3 coins based on input. [For a test case, this number (1, 2 or 3) is fixed.]

EXPLANATION:

Let $dp[n][c]$ denote the number of ways of distributing n coins to c charities. Then, a recursion relation can be derived in each of these cases:

At least 1 coin each:

$dp[n][c] = c * dp[n-1][c] + dp[n-1][c-1]$

Where, the first term denotes the number of ways of distributing $n-1$ coins to c charities and then distribute the n^{th} to any one, which can be done in c ways. The second term denotes the number of ways of distributing $n-1$ coins to $c-1$ charities and the n^{th} coin is given to the c^{th} charity.
Similarly, we get the following recurrence relations in the other two cases:

At least 2 coins each:

$dp[n][c] = c * dp[n-1][c] + (n-1) * dp[n-2][c-1]$

At least 3 coins each:

$dp[n][c] = c * dp[n-1][c] + \binom{n-1}{2} * dp[n-3][c-1]$

The final answer required in each case is $dp[N][C]$. As the number is huge, use modular arithmetic appropriately where required.

Time Complexity:

$O(T \times N \times C)$

PREREQUISITES:

Nim 120 , Mo's algorithm 137 , Basic Combinatorics

PROBLEM:

You are given an array A. You are given Q independent queries (L, R). For each query, a two-player game will be played on the array $A[L, R]$. The two players will alternate, with the first player starting. On each turn, a player has to remove a non-empty subset of elements from the array with the same value. The player who can't choose any elements loses. Find the number of subsets the first player can choose to win.

QUICK EXPLANATION:

This is a Nim game, where each distinct element in the array corresponds to a pile and the size of the pile is the number of occurrences of that element. Using Mo's algorithm, we can maintain the counts of elements in the subarrays for the queries. Let x be the xor of all counts. There are $O(\sqrt{N})$ distinct counts, so for each query, we can iterate through all distinct counts c. We check if $c \oplus x < c$, and if so, we add $\binom{c}{c \oplus x}$ to the answer.

EXPLANATION:

We should remap the values in A to values between 1 and N so that we can store counts of values in an array of size N.

Note that the game is really just a Nim game.

Explanation

- Both players alternate.
- The player who can't move loses.
- Each turn the player must remove at least one element.
- Each turn the player can only remove from one pile → Each turn the player can only remove one type of element.

A state is losing if the xor sum of all pile sizes is 0 (standard Nim fact). In our case, the each pile is the set of elements with the same value and the pile sizes are the counts of a certain element.

The first thing we need to do is to calculate the counts of values in the subarray. Let c_i be the number of times i appears in the subarray and let x be the xor sum of c_i for all valid i.

In order for the first player to win, they need to modify one of the pile sizes so that x becomes 0 (giving the second player a losing state).

Let's iterate over i and count the number of ways the first player can remove elements with value i to make $x = 0$. Let d be the new count of value i after we remove some elements. $x \oplus c_i \oplus d$ is the new xor sum of the pile sizes, and we need that to be 0. We can find d to be $x \oplus c_i$.

Since we are removing elements from the pile, we need $d < c_i$ to be satisfied, otherwise the number of ways to remove i and win is 0. How many ways are there if $d < c_i$? We are choosing d of the elements to keep (and removing the rest), so we should add $\binom{c_i}{d}$ to the answer.

In summary, a solution which gets the first two subtasks (30 points) is below:

- For each query:
- Find all c_i and calculate x.

- Iterate through all valid values i in $A[L, R]$:
  - If $(c_i \bigoplus x) < c_i$, add $\binom{c_i}{c_i \bigoplus x}$ to the answer.

The full solution is similar, but we need some tricks to speed up the current solution. The first obstacle is calculating c_i and x fast enough and the second obstacle is iterating through all i and calculating the answer fast enough.

Answering subarray queries related to the counts of values in the subarray is a standard Mo's Algorithm problem. We can process the queries in an order so that when we answer a query, the array c will be updated to represent the counts for the subarray. Maintaing x in this algorithm is not much harder.

Details

How do we overcome the second obstacle? Notice that if two piles have the same size, then their answer will be the same. Let cc_i be the number of j such that $c_j = i$ (count array of the count array c). While finding the answer, we will only iterate through distinct counts j and multiply the answer for the count by cc_j.

Notice that the sum of all counts $\leq N$, so the number of distinct counts is $O(\sqrt{N})$ (sum of first $O(\sqrt{N})$ positive integers is $O(N)$). If we can somehow maintain cc and the set of distinct counts for each query, we can answer each query in $O(\sqrt{N})$.

cc is pretty simple to maintain in Mo's algorithm. We can maintain the set of distinct counts with a binary search tree (set in C++), but it adds an additional $O(\log N)$ factor which will cause our solution to TLE.

Instead, we will use a data structure which supports:

1. Add an element $0 \leq x \leq N$ in constant time (if the element already exists, nothing happens).
2. Remove an element $0 \leq x \leq N$ in constant time.
3. Iterate over all elements in the set in $O(s)$ time if s is the number of elements in the set.

The setter, tester, and I each have different solutions for this data structure.

Setter

Maintain a doubly linked list with the counts as the elements. This only works in this problem because the counts either increase or decrease by 1.

Tester

Create a bitset of size $N + 1$. Adding and removing elements correspond to setting and clearing bits (which are constant time). Iterating over the set bits in a bitset works in $O(s + N/64)$, which is good enough.

Editorialist

The first and third operations can be supported easily with a list (vector in C++). We will additionally store a boolean array `inlist[i]` which tells us if an element is in the list. When we add an element i to the set, we should check that `inlist[i]` is false first.

When we want to perform the second operation, we will do so "lazily": we won't actually do it, but the next time we iterate over the elements in the set, we will remove it if we find the it should have been removed (if $cc = 0$).

The pseudocode for the third operation is shown below:

```
newlist
for x in list:
    if x should have been removed
        inlist[x]=false
        continue
```

```
    //do whatever with x
    add x to newlist
list = newlist
```

Here is the final upd function when we add or remove elements in Mo's algorithm:

```
upd(i, y):
    x^=c[i]
    --cc[c[i]]
    if cc[c[i]]==0:
        remove c[i] from the set of distinct counts
    c[i]+=y
    x^=c[i]
    ++cc[c[i]]
    add c[i] to the set of distinct counts (if it is not already in the set)
```

Mo's algorithm runs in $O(N\sqrt{Q})$ and each query takes $O(\sqrt{N})$ time, so the final time complexity is $O(N\sqrt{Q} + Q\sqrt{N})$.

PREREQUISITES:

Modulo operator and Basic Combinatorics.

PROBLEM:

Given two integers N and P, suppose the maximum value of $(((N \bmod i) \bmod j) \bmod k) \bmod N$ be M where $i,j,k \in [1,P]$, Find the number of ways to select $i,j,k \in [1,P]$ such that $(((N \bmod i) \bmod j) \bmod k) \bmod N$ equals M.

SUPER QUICK EXPLANATION

- The maximum value of $N \bmod x$ where $x \in [1,N]$, if N is odd, is $(N-1)/2$ when $x = (N+1)/2$, and if N is even, is $N/2-1$ when $x = N/2+1$.
- We can achieve $(((N \bmod i) \bmod j) \bmod k) \bmod N = M$ in three ways. Let $x = \lceil (N+1)/2 \rceil$
  - $i = x$ and $j,k > M$.
  - $i > N, j = x$ and $k > M$.
  - $i,j > N$ and $k = x$.
    Each of this case can be easily computed.

EXPLANATION

First of all, Let is find this value M. It has to be less than $min(i,j,k,N)$ which implies, $M < N$. Hence, if we want $M > 0$, we need $(((N \bmod i) \bmod j) \bmod k) < N$. So, We know for sure, that to maximize M, $min(i,j,k) \leq N$. Hence, we need maximum $(((N \bmod i) \bmod j) \bmod k) < N$ and now we can ignore the last $\bmod N$.

So, The maximum $N \bmod x$ can attain is $\lfloor (N-1)/2 \rfloor$. This happens when $x = \lceil (N+1)/2 \rceil$. It can be easily verified either by checking by hand, or writing a simple program 😉

Now, try finding out number of ways $(((N \bmod i) \bmod j) \bmod k)$ equals M. It can be approached in Simple case base analysis.

We can try all possible triplets of (i,j,k) and generalize them into three cases.

- When $i = \lceil (N+1)/2 \rceil$ and $j,k > M$
- When $i > N, j = \lceil (N+1)/2 \rceil$ and $k > M$
- When $i,j > N$ and $k = \lceil (N+1)/2 \rceil$

In all three cases, we can simply count the number of triplets (i,j,k) satisfying any condition and print the answer.

Corner Case

When $N \leq 2$, $M = \lfloor (N-1)/2 \rfloor = 0$. This is because we cannot achieve $(((N \bmod i) \bmod j) \bmod k) \bmod N > 0$. So, all triplets (i,j,k) are valid.

Alternate solution - read at your own risk, you have been warned 😃

For those curious enough not to be satisfied with such solutions, there also exists a pattern based solution too, using basic math. Just use brute solution to find the first terms of series and solve using the pattern formed. Number 6 is important. Enjoy 😃

Time Complexity

Time complexity is $O(1)$ per test case.

PREREQUISITES:

Nim 120 , Mo's algorithm 137 , Basic Combinatorics

PROBLEM:

You are given an array A. You are given Q independent queries (L, R). For each query, a two-player game will be played on the array $A[L, R]$. The two players will alternate, with the first player starting. On each turn, a player has to remove a non-empty subset of elements from the array with the same value. The player who can't choose any elements loses. Find the number of subsets the first player can choose to win.

QUICK EXPLANATION:

This is a Nim game, where each distinct element in the array corresponds to a pile and the size of the pile is the number of occurrences of that element. Using Mo's algorithm, we can maintain the counts of elements in the subarrays for the queries. Let x be the xor of all counts. There are $O(\sqrt{N})$ distinct counts, so for each query, we can iterate through all distinct counts c. We check if $c \oplus x < c$, and if so, we add $\binom{c}{c \oplus x}$ to the answer.

EXPLANATION:

We should remap the values in A to values between 1 and N so that we can store counts of values in an array of size N.

Note that the game is really just a Nim game.

Explanation

- Both players alternate.
- The player who can't move loses.
- Each turn the player must remove at least one element.
- Each turn the player can only remove from one pile → Each turn the player can only remove one type of element.

A state is losing if the xor sum of all pile sizes is 0 (standard Nim fact). In our case, the each pile is the set of elements with the same value and the pile sizes are the counts of a certain element.

The first thing we need to do is to calculate the counts of values in the subarray. Let c_i be the number of times i appears in the subarray and let x be the xor sum of c_i for all valid i.

In order for the first player to win, they need to modify one of the pile sizes so that x becomes 0 (giving the second player a losing state).

Let's iterate over i and count the number of ways the first player can remove elements with value i to make $x = 0$. Let d be the new count of value i after we remove some elements. $x \oplus c_i \oplus d$ is the new xor sum of the pile sizes, and we need that to be 0. We can find d to be $x \oplus c_i$.

Since we are removing elements from the pile, we need $d < c_i$ to be satisfied, otherwise the number of ways to remove i and win is 0. How many ways are there if $d < c_i$? We are choosing d of the elements to keep (and removing the rest), so we should add $\binom{c_i}{d}$ to the answer.

In summary, a solution which gets the first two subtasks (30 points) is below:

- For each query:
- Find all c_i and calculate x.

- Iterate through all valid values i in $A[L, R]$:
  - If $(c_i \bigoplus x) < c_i$, add $\binom{c_i}{c_i \bigoplus x}$ to the answer.

The full solution is similar, but we need some tricks to speed up the current solution. The first obstacle is calculating c_i and x fast enough and the second obstacle is iterating through all i and calculating the answer fast enough.

Answering subarray queries related to the counts of values in the subarray is a standard Mo's Algorithm problem. We can process the queries in an order so that when we answer a query, the array c will be updated to represent the counts for the subarray. Maintaing x in this algorithm is not much harder.

Details

While performing Mo's algorithm, when we need to add an element at index i to the subarray, we call $upd(i, 1)$. When we need to remove an element at index i from the subarray, we call $upd(i, -1)$.

```
upd(i, y):
    x^=c[i]
    c[i]+=y
    x^=c[i]
```

How do we overcome the second obstacle? Notice that if two piles have the same size, then their answer will be the same. Let cc_i be the number of j such that $c_j = i$ (count array of the count array c). While finding the answer, we will only iterate through distinct counts j and multiply the answer for the count by cc_j.

Notice that the sum of all counts $\leq N$, so the number of distinct counts is $O(\sqrt{N})$ (sum of first $O(\sqrt{N})$ positive integers is $O(N)$). If we can somehow maintain cc and the set of distinct counts for each query, we can answer each query in $O(\sqrt{N})$.

cc is pretty simple to maintain in Mo's algorithm. We can maintain the set of distinct counts with a binary search tree (set in C++), but it adds an additional $O(\log N)$ factor which will cause our solution to TLE.

Instead, we will use a data structure which supports:

1. Add an element $0 \leq x \leq N$ in constant time (if the element already exists, nothing happens).
2. Remove an element $0 \leq x \leq N$ in constant time.
3. Iterate over all elements in the set in $O(s)$ time if s is the number of elements in the set.

The setter, tester, and I each have different solutions for this data structure.

Setter

Maintain a doubly linked list with the counts as the elements. This only works in this problem because the counts either increase or decrease by 1.

Tester

Create a bitset of size $N + 1$. Adding and removing elements correspond to setting and clearing bits (which are constant time). Iterating over the set bits in a bitset works in $O(s + N/64)$, which is good enough.

Editorialist

The first and third operations can be supported easily with a list (vector in C++). We will additionally store a boolean array `inlist[i]` which tells us if an element is in the list. When we add an element i to the set, we should check that `inlist[i]` is false first.

When we want to perform the second operation, we will do so “lazily”: we won’t actually do it, but the next time we iterate over the elements in the set, we will remove it if we find the it should have been removed (if $cc = 0$).

The pseudocode for the third operation is shown below:

```
newlist
for x in list:
    if x should have been removed
        inlist[x]=false
        continue
    //do whatever with x
    add x to newlist
list = newlist
```

Here is the final upd function when we add or remove elements in Mo’s algorithm:

```
upd(i, y):
    x^=c[i]
    --cc[c[i]]
    if cc[c[i]]==0:
        remove c[i] from the set of distinct counts
    c[i]+=y
    x^=c[i]
    ++cc[c[i]]
    add c[i] to the set of distinct counts (if it is not already in the set)
```

Mo’s algorithm runs in $O(N\sqrt{Q})$ and each query takes $O(\sqrt{N})$ time, so the final time complexity is $O(N\sqrt{Q} + Q\sqrt{N})$.

PREREQUISITES:

Dynamic Programming, Matrix Exponentiation, Maths.

PROBLEM:

Considering there are N people participating and there are M problems in the contest. What would be the expected number of problems solved by the person at rank K on the leaderboard?

If the expected value is P/Q you must print the value $P * Q^{-1}\%MOD$, where $MOD = 10^9 + 7$.

EXPLANATION

Consider the leaderboard as a matrix of size $N * M$.

We define $f(n, m)$ as the number of total configurations that are possible if there are n participants and m problems.

$f(n, m) = \sum_{i=0}^{m} \binom{m}{x} * f(n-1, i)$

This can be calculated in $O(m^3 log_2(n))$ using matrix exponentiation :-

$[f_{n-1}(0) \quad f_{n-1}(1) \quad . \quad . \quad . \quad f_{n-1}(m)] * M = [f_n(0) \quad f_n(1) \quad . \quad . \quad . \quad f_n(m)]$
where, M is $(m+1) * (m+1)$ matrix, such that -

$$M = \begin{bmatrix} \binom{m}{0} & \binom{m}{0} & . & . & \binom{m}{0} \\ 0 & \binom{m}{1} & . & . & \binom{m}{1} \\ 0 & 0 & . & . & \binom{m}{2} \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & . & . & \binom{m}{m} \end{bmatrix}$$

Therefore $F_n = F_0 * M^n$ and $F_n(i)$ is the number of configurations such no one solved more than i problems.

and $F_0 = [1 \quad 1 \quad . \quad . \quad . \quad 1]$

We define $g(n, m, k)$ as the number of total configurations that are possible if there are n participants and m problems and the last participant must solve at least k problems.

$g(n, m, k) = F_0^{'} * M^n$, where

$F_0^{'} = [0 \quad 0 \quad . \quad 1 \quad 1 \quad . \quad 1]$ such that for all $i \geq k$, $F_0^{'}(i) = 1$.

We define $h(n, m, k)$ as the number of total configurations that are possible if there are n participants and m problems and the first participant must solve at most k problems.

$F_n = F_0 * M^n$, and -

$h(n, m, k) = F_n(k)$

Now, the answer to the problem is the expected value of the number of problems solved by person k, which can be calculated for each i where i is the number of problems solved by a person at rank k as -

$answer = \sum_{i=0}^{m} p(k,i) * i$

where $p(i) =$ the probability of solving i problems by the person at rank k.

and $p(k,i) = \binom{m}{i} * g(k-1,m,i) * h(n-k,m,i)/f(n,m)$

i.e. product of all the possibilities of choosing i problems for rank k , number of ways such that the top $k-1$ people solve at least i problems and number of ways such that the bottom $n-k$ people solve at most i problems.

Therefore the final answer will be,

$answer = f(n,m)^{-1} * \sum_{i=0}^{m} \binom{m}{i} * g(k-1,m,i) * h(n-k,m,i) * i$

Note - g and h can be calculated in $O(n^2)$ by precomputing the matrix M for each of them separately.

So, the total time complexity will be $O(M^3 log_2(N) * T)$.

Time Complexity: $O(T * M^3 * log_2(N))$
Space Complexity: O(M^3)
Problem Setter Code: Solution 17

PREREQUISITES:

Sorting, combinatorics

PROBLEM:

The **strength** of an array $a = (a_1, \ldots, a_n)$ is the sum of $|i - j|$ for all $1 \leq i \leq j \leq n$ such that $a_i = a_j$.

How many arrangements of a are there such that the strength is the maximum possible among all rearrangements? (modulo $10^9 + 7$)

QUICK EXPLANATION:

Let v be a value of a which appears k times. Replace the i th occurrence of v with $2i - k - 1$. Do this for all distinct values v.

Let c_w be the number of times w appears in this new array. The answer is then

$$\prod_{w=-(n-1)}^{n-1} c_w!$$

EXPLANATION:

The first observation is that the actual values of a themselves don't matter a lot. The positions of a particular value doesn't matter either, because we're going to rearrange a anyway. For a given value v of a, all that matters really is **the number of times v appears in a**, because all rearrangements of a will contain that many appearances of v.

Let's fix a value v, and say it appears k times in a. Obviously, the best way to place these v s in the rearrangement is to place (roughly) half in one end and the remaining ones in the other. For example, if the value 5 appears 6 times, then placing it in the following manner maximizes its contribution to the strength: $(5, 5, 5, ?, ?, ?, ?, ?, 5, 5, 5)$.

However, when there are other values involved, things are trickier. It's tempting to say that the best way would be to place the most frequent values outside the array, but actually we can't be greedy like this. It's true that the more times a number appears, the higher its contribution to the strength is, but when combined with other values, it's not always optimal to place them in the outer edges of the array. For example, if $a = (4, 4, 4, 4, 5, 5, 5, 5, 5, 5)$, then our method will yield $(5, 5, 5, 4, 4, 4, 4, 5, 5, 5)$, but this is not the best; the rearrangement $(5, 4, 5, 4, 5, 5, 4, 5, 4, 5)$ yields a higher strength.

To find the optimal arrangement, we need to dig a little deeper. Again let's look at a particular value v which appears k times. Suppose $p_1 < p_2 < \ldots < p_k$ are the positions of the v s. Then the contribution of the v s to the strength is:

$$\begin{aligned}\sum_{1\le i\le j\le k} |p_i - p_j| &= \sum_{1\le i\le j\le k} (p_j - p_i) \\ &= \sum_{1\le i\le j\le k} p_j - \sum_{1\le i\le j\le k} p_i \\ &= \sum_{1\le j\le k} p_j \cdot j - \sum_{1\le i\le k} p_i \cdot (k-i+1) \\ &= \sum_{1\le i\le k} p_i \cdot i - \sum_{1\le i\le k} p_i \cdot (k-i+1) \\ &= \sum_{1\le i\le k} p_i \cdot (2i-k-1)\end{aligned}$$

We now notice the following: the contribution of each position p_i is equal to itself **weighted by** $(2i-k-1)$. For other values than v, a similar situation happens, though the weights may be different. Thus, we can say the following:

The contribution of the i th occurrence of v is its position weighted by $(2i-k-1)$, where k is the number of times v appears.

Let's consider the *weight array* of a, i.e. the sequence of weights of all numbers. For example, for the array

$$a = (1, 6, 9, 1, 1, 6, 7, 1, 7, 7),$$

the weight array is

$$(-3, -1, 0, -1, 1, 1, -2, 3, 0, 2).$$

(For example, check that the four instances of 1 have weights -3, -1, 1 and 3.)

Now, assuming we can rearrange this array any way we want, what is the rearrangement of this weight array that maximizes the strength? Clearly we can be greedy with this and the best way is simply to *sort* the array, yielding: $(-3, -2, -1, -1, 0, 0, 1, 1, 2, 3)$. Thus, we just found a very fast way of computing the maximum strength!

There's a slight issue with this though. Clearly this is the best we can do, but note that we assumed we can rearrange the array any way we want, but this is not true. For example, we can't place the second occurrence of a value before its first occurrence. But since the weight of the i th occurrence is strictly less than the $(i+1)$th occurrence of a given value, any arrangement placing these occurrences out of order will never yield the maximum strength, so we're fine 🙂

The remaining part is to count how many arrangements yield this maximum. But this isn't too hard. First, count the number of times each weight appears. Now we can't reorder the weight array because we have to keep them sorted, but same-weight entries can be arranged amongst themselves. For any weight w appearing c_w times, there are $c_w!$ ways to arrange these values, so we determine that the number of arrangements is simply

$$\prod_{w=-(n-1)}^{n-1} c_w!$$

(Notice that all weights are $> -n$ and $< n$)

The running time is $O(n \log n)$, dominated by sorting / counting frequencies of the a_is. All other parts of the algorithm can be done in $O(n)$.

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

Time Complexity:

$O(n \log n)$

PREREQUISITES:

Graph techniques like DFS, BFS and DSU [1], Basic Combinatorics, Modular Arithmetic and Modular Inverse [1].

PROBLEM:

Given a grid A of size $N \times M$ (N rows and M columns) and an integer K ($K \leq N.M$), find the smallest *connected region* with size $S \geq K$ and compute $\binom{S}{K} \bmod (10^9 + 7)$. If no such region exists, print -1.

Note: If we imagine this grid as an undirected graph where each cell represents a vertex. There is an edge between two vertices if they have the same value ($A_{i,j} = A_{p,q}$) and they are neighbors (each cell has at most 8 neighbors, well mentioned in the problem statement) to each other. A *connected region* is a connected component [1] in this graph.

EXPLANATION:

First of all we should figure out the size of the region which will minimize the number of choices. Using basic combinatorics, we can figure that out very easily. We know,

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{\prod_{i=1}^{r}(n-r+i)}{r!}$$

In our problem, K (r in the above formula) is fixed for every choice. So, in order to minimize the number of choices, we need to minimize the numerator in the above formula. So, we need to choose the size S ($S \geq K$) (n in the above formula) as small as possible which will minimize the numerator in the above formula and hence minimize the number of choices.

Now we can think about the strategy to solve this problem programmatically. We need to find the connected component in this graph with the smallest size $\geq K$. This can be done easily using standard graph traversal techniques like DFS or BFS or by using data structures like DSU. Representing this grid as a graph is also a trivial task as the concepts of neighbors and connected regions are well defined in the problem statement itself.

Factorials and their respective modular multiplicative inverses under modulo $10^9 + 7$ (which is a prime number) can be pre-computed easily using basic concepts of Modular Arithmetic.
(For calculating modular multiplicative inverses, we can use Fermat's little theorem [1] which works in logarithmic time due to Modular Exponentiation for a single inverse. In the solution mentioned below, modular inverse has been calculated in constant time for a single inverse, which is faster but not necessarily needed.)

SOLUTION:

Setter's Solution

COMPLEXITY:

Time complexity: $O(N \times M)$ per test.
Space Complexity: $O(N \times M)$ auxiliary space per test.

[Pre-processing takes $O(C)$ time with $O(C)$ auxiliary space where $C \leq 10^5$]

Feel free to share your approach. If you have any queries, they are always welcome.

PREREQUISITES:

Basic Combinatorics, Probability, Modular Multiplicative Inverse 1

PROBLEM:

You need to calculate the probability of making team of K fighters from N fighters on given condition(Fighters from Universe 7 > Fighters from Universe 6) by asking 2 types of queries,

- Type 1 : At most $n-1$ times, Query is defined as, for i & j grader replies with if both fighters are from same universe or not
- Type 2 : Only once,Query is defined as, for any index i grader replies with the Universe of i^{th} fighter

The probability can be represented as P/Q you have to calculate $P * Q^{-1} \ mod \ (10^9+7)$, Where Q^{-1} is the modular multiplicative inverse of Q $(mod \ (10^9+7))$

QUICK EXPLANATION:

- Ask Type 2 query to get the universe of one fighter.
- Based on the already found universe of the fighter we can get the Universe other fighters from Type 1 Query by asking it exactly $N-1$ times from which we can use counts of the universes.
- Find the probability of having fighters from Universe 7 > fighters from Universe 6 in the team of K fighters from total N fighters

EXPLANATION:

About Interaction, to find probability we need Counts of fighters from each universe.
Consider the count of fighters from Universe 7 as $uni7$ and count of fighters from universe 6 as $uni6$.

- We know that the second type of query gives the Universe of the fighter,so we can find the universe of fighter 1 by giving query as $F \ 1$ and increase the count of the given universe $i.e$ if the Universe is 7 then increase $uni7$ and vice-versa for $uni6$.
- Now we can ask the Type 1 queries as $Q \ 1 \ i \ (2 \leq i \leq N)$ if we get response as same universe we increase the count of the universe of 1^{st} fighter, if not we increase the count of the inverse universe of 1^{st} fighter.

After we have processed all queries we have $uni7$ and $uni6$.

Now we have to calculate probability.
We need to select K fighter from total N fighters so we can write denominator Q = $\binom{n}{k}$

Calculation of Numerator:

Consider we select i fighters from $uni7$ so we have to select $k-i$ fighters from $uni6$.
So we have to select upper and lower bound of i to calculate the probability.

Calculation of Lower Bound
We want $uni7 > uni6$ So we can compute lower bound like below.

$$i > k - i$$
$$2 * i > k$$

$i > k/2$

From above proof we can choose i from $k/2 + 1$, but what if we don't have enough fighters from $uni6$ to satisfy $k - i$?
So, we can define the lower bound as $max(floor(k/2) + 1, k - uni6)$.

Calculation of Upper Bound
As we want at least 1 fighter from each Universe we can go upto $k - 1$, But what if we don't have enough fighters from $uni7$ to get i fighters from $uni7$.

So, we can define the lower bound as $min(uni7, k - 1)$.
From Above proofs we can define the numerator.

$P = \sum_{i=max(int(k/2)+1, k-uni6)}^{min(uni7, k-1)} \binom{uni7}{i} * \binom{uni6}{k-i} \bmod m$ where $m = 10^9 + 7$.

We have P and Q and the required probability is in the form of $P * Q^{-1} \ mod \ (10^9 + 7)$.

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

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

Implementation Tips

We got the probability,

We are multiplying every combination to calculate P for that we can use Modular Arithmatic,

In Modular Arithmtic $(a * b) \bmod m = ((a \bmod m) * (b \bmod m)) \bmod m$
So instead of calculating whole we can use mod at every step.

We know that $\binom{n}{r} = \frac{n!}{r!(n-r)!}$, to calculate the factorials every time is costly and the constraints are so big $(2 \leq N \leq 10^5)$ so factorials of these numbers will also be big.

As we are doing multiplication with $\bmod$ at each step(described above), We can store factorials with mod.
$factorial[i] = (i * factorial[i - 1]) \bmod m$
To Calculate $\binom{n}{r} = \frac{n!}{r!(n-r)!}$, We need $factorial[n], factorial[r], factorial[n - r]$

But,

$(a/b) \bmod m \neq ((a \bmod m)/(b \bmod m)) \bmod m$.

So, to divide with $\bmod$, Since we have $m = 10^9 + 7$ as prime we can apply Fermat's little theorem.
We can define $\binom{n}{r} = factorial[n] * modInverse(factorial[r]) * modInverse(factorial[n - r])$,

Where modInverse() method contains Fermat's little theorem.

We can precompute modInverse of all factorial possible values.(so we don't need to find multiplicative inverse everytime)
Therefore, when we want $\binom{n}{r}$ we can get it in $O(1)$ time.

TIME COMPLEXITY:

Time complexity is $O(N)$ for each test case. (Ignoring the precomputation)
Precomputation can be done in $O(100000 * log(m))$

PREREQUISITES:

Catalan Number and their Results.

PROBLEM:

Sokka has to travel on a square grid of NxN from (0,0) to (N-1,N-1) and has to pay money each time he makes a transition (refer the problem statement for knowing what a transition is) and moving only right or below from the current cell in the grid.

QUICK EXPLANATION:

If you observe carefully , the problem can be reduced to finding number of paths in a grid from one corner to diagonally opposite corner which cross the diagonal.

EXPLANATION:

We are asked the Probability that the coins are lesser than the initial value when the journey ends. If you have understood the basic question , we have to pay a coin each time the path crosses the diagonal. So we have to calculate the probability of the paths which crosses the diagonal of the grid, no matter how many times, because in all the paths which crosses the diagonal we will have to pay atleast one coin and the money in the bag after the journey will be less.

Now let's define n=N-1. In a NxN grid if we have to move from (0,0) to (N-1,N-1) following the rules as described in the problem , we need to take n Right(R) moves and n Down(D) moves. Total number of distinct possible paths will be T=2nCn(number of permutations of a string of size 2n with n 'R' characters and n 'D' characters. If we subtract the number of paths which never cross diagonal from this total value we will get our answer i.e. the number of paths which cross the diagonal.

If you observe, a path which doesn't cross the diagonal is either always below the diagonal or it is always above it. Let's find out the number of paths which are always below diagonal.

If we define our path as a sequence of D and R moves , a path which is always below diagonal will be a path with the following property:
If you traverse the path sequence from left to right (1 to 2n) , at any point the number of D's encountered till that point must be greater than or equal to the number of R's encountered upto that point. If you treat D as '(' and R as ')' , the problem reduces to number of valid paranthesizations with n pairs of opening and closing braces. So, our answer is nothing but the nth Catalan number. Similarly , number of paths which are always above the diagonal will be equal(Due to symmetricity or just replace D and R in the above proof).

Thus, the number of paths which never cross the diagonal are 2x(nth Catalan Number).

So, the number of paths which cross the diagonal will be (2nCn)-2*((1/(n+1))2nCn), which will be equal to P=((n-1)/(n+1))2nCn. Thus the probability is P/T=(n-1)/(n+1) where n=N-1.

The initial value of coins in the bag 2*N is useless extra information as the maximum number of times Sokka makes a transition(or cross the diagonal) will be around half of this value, so any path is possible.

PREREQUISITES:

dynamic programming, simple combinatorics.

PROBLEM:

Given an array A. You have to find number of sub-sequences of A which are not arithmetic progressions(AP).

EXPLANATION

As total number of sub-sequences in the array A will be 2^n (including the empty sequence),
Excluding the empty sequence, we will have $2^n - 1$ sub-sequences.

So instead of finding number of sequences which are not arithmetic progressions, we will find number of sequences which are
arithmetic progressions and subtract it from $2^n - 1$.

Terminology

AP is defined as a series of a, a + d, a + 2 * d, etc.
By difference of AP we mean d.

Number of APs in an array A

Notice the following the observation.

- Elements of the array lie between 1 to 100 inclusive.
- Due to previous condition, maximum difference of AP can be at most 100.
- Similarly, minimum difference of AP can be at most -100.

So the values of elements of the array are small. We will make use of this observation.
Let us say we iterate over the difference of the AP and try to count number of APs in the array having the given difference.

Number of APs having difference d in an array A

We will solve this problem by dynamic programming. Let dp[i] denote the number of AP's ending at i and having difference equal to d.
So if our current number is equal to A[i], we need to
find all the positions j < i such that A[j] = A[i] - diff and we will take sum of dp[j] for those j's. It is equivalent to extending the
APs ending at position j with the element at position i.

Hence *dp[i] = 1 + sum_(j = 1 to i - 1) dp[j] such that A[j] = A[i] - diff*

We are adding 1 for the case when our current number is the only number in the AP. (1 element is also an AP according to problem statement.)

Naively computing the above recurrence will take time equal to $O(N^2)$ which is not going to pass in the given time.
We need to optimize this.

Note that we can optimize this by maintaining a sum array where sum[x] will record sum of all the dp's where the value of array elements was x.
Mathematically speaking, sum[x] = sum of all dp[i]'s such that A[i] = x.

After this, our dp will be.

Hence dp[i] = sum[A[i] - diff] + 1.

Now with this optimization, our dp computation will take O(N) time.

Note that we are doing slight over-counting in the APs of just one element. Each iteration of diff will count A element APs, Hence it will
amount to over-counting. For that we can simply remove n single elements APs in the iteration over diff variable. In the end, we can simply
add the n 1 element APs.

PREREQUISITES:

Combinatorics, Observation

PROBLEM:

Given a complete binary tree with depth D and $N = 2^{D+1} - 1$ nodes, rooted at node 1. The edges connecting nodes at Odd depth to their parent are colored white, while edges connecting nodes at even depth with their parent are colored black.

A strip is a cycle where no two adjacent edges have the same color.

When choosing two distinct nodes u and v, and color c randomly equiprobably, and adding an edge from u to v with color c, find the probability of making a strip.

QUICK EXPLANATION

- A strip can be generated only when the edge added from a node to its ancestor.
- The strip always has even length, so we can connect a node with its ancestors at an odd distance to make a strip.
- Now we just need to keep track of the number of ancestors of a node at both odd and even distance.

EXPLANATION

Let us add nodes depth-wise and keep track of the number of good tuples (u, v, c) which generate a strip.

Lemma 1: A strip can be generated only when the edge added from a node to its ancestor.
Proof: Let's assume a strip is generated by connecting two nodes A and B where the lowest common ancestor of A and B is C. Now, consider both children of node C. They must lie on the strip, and the edges connecting them to C are of the same color and are adjacent. This contradicts with our definition of the strip. Hence, a strip can only be generated when a node is connected to its ancestor.

Lemma 2: Node A and its ancestor node B form a strip if and only if A and B have an odd number of edges between them.
Proof: Let's assume the distance between A and B is even. This way, the edge connecting A to its parent and the edge connecting child of B to B have the opposite color. Hence, we cannot choose any valid color for the new edge without violating strip property.

The above two lemmas give us all the information we need to count the number of good tuples (u, v, c) which generates a strip.

Consider all nodes at depth d. Say there are P such nodes, C_0 denote the number of ancestors at even depth and C_1 denote number of ancestors at odd depth. The number of valid pairs increases by $2 * C_1$ if the current depth is even, otherwise by $2 * C_0$. For each pair of nodes, the color is automatically decided. The 2 factor appears since (u, v) is different from (v, u)

The total number of ways to choose tuples is $2 * N * (N - 1)$ where N is the number of nodes. Take care of the modulo.

We can also precompute the answer in advance since it only depends upon depth D

TIME COMPLEXITY

The time complexity is $O(D)$ per test case if not precomputing.

PROBLEM:

Given an array W of length N, find the expected score from the following process:

- Choose an integer k between 1 and N uniformly randomly.
- Let A be k not necessarily distinct elements chosen uniformly randomly and independently from W.
- Let B be chosen in the same way as A.
- The score is $\sum_{i=1}^{k} \sum_{j=1}^{k} \gcd(A_i, B_j)$.

QUICK EXPLANATION:

The score is just k^2 times the expected value of $\gcd(W_i, W_j)$, so the answer is just the expected value of k^2 times the expected value of $\gcd(W_i, W_j)$. To find the expected value of $\gcd(W_i, W_j)$, we will iterate over all g and count the number of pairs such that $\gcd(W_i, W_j) = g$.

EXPLANATION:

Note that the score is just k^2 times the expected value of $\gcd(W_i, W_j)$, where i and j are chosen uniformly randomly.

Proof

Since k is independent from $\gcd(W_i, W_j)$, the answer is the expected value of k^2 times the expected value of $\gcd(W_i, W_j)$.

We can easily calculate the expected value of k^2.

Explanation

Add up k^2 for k from 1 to N and divide by N. Alternatively, use the well-known formula for the sum of the first k squares, which is $\frac{k(k+1)(2k+1)}{6}$.

What remains is to calculate the expected value of $\gcd(W_i, W_j)$, which is the same as the sum of all $\gcd(W_i, W_j)$ divided by N^2.

Finding the sum of all $\gcd(W_i, W_j)$ is a well-known problem.

Explanation

First, we will calculate a frequency array c of all the elements in W. Then, we will calculate an array d such that d_i is the number of elements in W which are multiples of i. The pseudocode is shown below:

```
for i from 1 to MAX:
    j = i
    while j <= MAX:
        d[i] += c[j]
        j += i
```

The total time can be approximated by $\sum_{i=1}^{N} \frac{N}{i}$, which is well-known to be equal to $O(N \log N)$ (approximate the sum with an integral).

Next, set d_i to d_i^2, so d_i now represents the number of pairs of elements in W such that both elements of the pair are multiples of i.

Note that d_i also represents the number of pairs such that the gcd of the first element and the second element is a multiple of i. Next, we want to find e_i, which is the number of pairs such that the gcd of the first element and the second element is exactly i. We can find e_i with complementary counting. We subtract all cases when the gcd isn't exactly i but is a multiple of i:

```
for i from MAX to 1:
    e[i] = d[i]*d[i]
    j = 2*i
    while j <= MAX:
        e[i] -= e[j]
        j += i
```

The final sum of the gcd over all pairs can be found by summing up $i \cdot e_i$.

The total time complexity is $O(N \log N)$, where N also represents the maximum value in W.

PRE-REQUISITES

Math, Combinatorics, Bit Manipulation, Meet in the middle

PROBLEM

Given two integers N and B, you need to find the number of sequences of length N such that the beauty values of it's subsequences is exactly B. (B is actually B modulo **10^9 + 7**). Since, the number of such sequences can be fairly large, you have to calculate the answer modulo **10^9 + 7**. Beauty of a subsequence is defined as the product of length of the subsequence and the XOR of the numbers contained in it.

EXPLANATION

The brute approach here is pretty much easy to visualize. For any given N, there will be $(C-1)^N$ such subsequences. But, we need to find out which of the have the sum of the beauty values of their subsequence as B (modulo **10^9 + 7**). Fixing each such sequence, you calculate the sum of their beauty values of subsequence modulo **10^9 + 7**. In case, it equals B, you increment your answer by 1. So, your complexity will be huge in this case. Something like $O((C-1)^N * X)$ where X is the complexity to calculate the sum of beauty values of all subsequence in a particular sequence. This actually leads us no where close to achieving the intended solution.

The other way to think in the problems involving Bitwise operations is to visualize everything in terms of binary. So, in case if you have some fixed sequence, how do you calculate the sum of the beauty values of it's subsequences? Let us write down each number of the sequence in it's binary representation. Since, each number will be $lt; 2^{20}$, we can represent each of them in 20 binary bits to maintain consistency.

Claim 1: We can actually calculate the beauty value for each bit individually and then finally sum up the beauty values of all 20 bits to get the final value.

After representing a given list of numbers in binary, their Bitwise XOR is calculated as the summation of the $2^{i^{th} bit}$ when the i^{th} column has odd number of ones. What actually matters is the odd number of ones in the column. So, for a particular column which will have some **N** binary bits, we can calculate the sum of the beauty values of the subsequences just for that column.

Let n = number of ones and m = number of zeros in any particular column such that $n + m = N$

The column k will contribute value 2^k only when it has odd number of ones irrespective of any number of zeros contained in it. Having said that, let us select a subsequence of $(2i+1)$ ones from the total number of ones and j zeros from the total number of zeros in the k^{th} column. Number of ways to select $(2 * i + 1)$ ones from n ones will be $\binom{n}{2i+1}$ and similarly $\binom{m}{j}$ will be the number of ways to select j zeros from m zeros. In this case, total length of the subsequence will be $(2 * i + j + 1)$ and the XOR value will obviously be 2^k. Here, $(2i+1) \leq n$ which means $i \leq (n-1)/2$. Let $p = (n-1)/2$ so that $i \leq p$

With this, the sum of the beauty values contributed by all subsequence in k^{th} column becomes…

$$2^k * [\sum_{i=0}^{p} \sum_{j=0}^{m} (2i + j + 1) \cdot \binom{n}{2i+1} \cdot \binom{m}{j}]$$

Breaking this into two separate parts,

$$\implies 2^k * [\sum_{i=0}^{p}\sum_{j=0}^{m}(2i+1)\cdot\binom{n}{2i+1}\cdot\binom{m}{j} + \sum_{i=0}^{p}\sum_{j=0}^{m}(j)\cdot\binom{n}{2i+1}\cdot\binom{m}{j}]$$

$$\implies 2^k * [\sum_{i=0}^{p}(2i+1)\cdot\binom{n}{2i+1}\cdot(2^m) + \sum_{i=0}^{p}\binom{n}{2i+1}\cdot(m)\cdot(2^{m-1})]$$

$$\implies 2^k * [(n)\cdot 2^{n-2}\cdot(2^m) + 2^{n-1}\cdot(m)\cdot(2^{m-1})]$$

If you observe carefully, $(n)\cdot 2^{n-2}$ becomes undefined when $n \leq 1$. So, in case $n > 1$, the above equation can be simplified further as solved below.

$$2^k * [n\cdot 2^{n+m-2} + m\cdot 2^{n+m-2}]$$

$$2^k * 2^{n+m-2}\cdot(n+m)$$

$$2^{k+n+m-2}\cdot(n+m)$$

Since, $n+m=N$, the equation is simplified to …

$$N\cdot 2^{k+N-2}$$

Similarly, for the case when $n=1$, the equation is simplified to …

$$(N+1)*2^{N+M-2}$$

Well, for the case when $n=0$, which means the number of ones in the column will be 0, the value contributed will anyway be 0.

Let the value contributed by any bit column k to the sum of beauty of all subsequences be $F(k)$

$$F(k)=0,\ if\ n\ =\ 0$$

$$F(k)=(N+1)*2^{N+k-2},\ if\ n\ =\ 1$$

$$F(k)=N*2^{N+k-2}, if\ n\ >\ 1$$

The sum of the beauty values contributed by all such columns for a particular sequence will be :-

$$\sum_{k=0}^{19}F(k)$$

Each column can have only one of the three possible values.

- When count of ones = 0 then the number of such possible columns = 1
- When count of ones = 1, then the number of such possible columns = $\binom{N}{1}$
- When count of ones > 1, then the number of such possible columns = $2^N - 1 - \binom{N}{1}$

The first thing that comes to our mind is recurring by keeping the function $f(idx, beautyValue, sequences)$. Let us have a look at the pseudo code to understand it better. Let us denote the number of columns calculated above by $type[i]$ where $type[i]$ denotes the number of possible columns when there are i ones.

```
f(idx, beauty_val, sequences)
{
     if ( idx == 20 ) {
         if ( beauty_val == B ) return sequences;
         return 0;
     }
     ans = 0
     // Number of ones are zero
     ans += f(idx + 1, beauty_val, sequences)
     //Number of ones are 1
     ans += f(idx + 1, (beauty_val + F(idx))%MOD, sequences * type[1])
    //Number of ones are > 1
     ans += f(idx + 1, (beauty_val + F(idx))%MOD, sequences * type[2])
     return ans
}
```

We are very close to the actual solution now. But, the above implementation still means the complexity of order $O(3^{20})$.
How can we improvise? Can we apply meet in the middle technique? Yes. That is it. You can divide the columns into two parts.
So, let us say $Map(0, B1)$ will store the number of sequences having sum of beauty values of subsequences for first 10 columns i.e $[0..9]$ as $B1$. Similarly, $Map(0, B2)$ will store the number of sequences having sum of beauty values of subsequences for last 10 columns i.e $[10...19]$. Now, you can iterate over all distinct beauty values in $Map(0)$ and find the counter beauty values in $Map(1)$ such that beauty values in both sum up to B modulo MOD.

The overall complexity is now reduced to $O(3^{10} * log(3^{10}))$

For more details on the implementation, have a look at the setter or tester's solution.

Note

Please notd that above calculations might be done without considering $MOD = 10^9 + 7$. Please do not forget to consider MOD while actually implementing the above approach.

PREREQUISITES:

Combinatorics, Inclusion-Exclusion, and Expectation.

PROBLEM:

Given N intervals $[L_i, R_i]$ for $1 \leq L_i \leq R_i \leq 10^5$, consider all possible ways of choosing N values, i-th value being chosen in i-th interval. Let A be the product of chosen values and B be the gcd of chosen values, find the expected value of A/B over all possible ways.

QUICK EXPLANATION

-

EXPLANATION

Let us solve an easier problem. Instead of the expected value of A/B where A is the product and B is the greatest common divisor, let us calculate the expected value of the greatest common divisor of chosen values.

Let us count the number of valid N-tuples such that their GCD is x. Let's call this g_x. Instead, we can easily count the N-tuples such that their GCD is a multiple of x, calling this f_x. This just means that all the values in tuple should be a multiple of x.

Let us try each x and for each x, we find the number of valid N-tuples which have all elements as multiple of x. In this tuple, the i-th value is between L_i and R_i and a multiple of x. There are only $\lfloor \frac{R_i}{x} \rfloor - \lfloor \frac{L_i - 1}{x} \rfloor$ possible choices for i-th value in tuple. Each value in tuple can be independently chosen from each other, so the number of valid tuples become $\prod (\lfloor \frac{R_i}{x} \rfloor - \lfloor \frac{L_i - 1}{x} \rfloor)$. Let's do this for each x, so this takes $O(N * MX)$ time. We'll optimize it later.

Now, for each x, we have found g_x for all values of x. We need to compute f_x from g_x which we can do in $O(N * log(N))$ as follows.

Pseudo Code

```
f[i] -> Number of tuples with GCD as multiple of x
g[i] -> Number of tuples with GCD as x
for(int i = MX; i >= 1; i--){
    g[i] = f[i]; //Number of tuples with GCD as multiple of i
    for(int j = 2*i; j <= MX; j += i){
        g[i] -= g[j]; //Subtracting tuples with GCD j > i such that i divides j
    }
}
```

Let's come back to the original problem. We want to find the expected value of the product of values in tuple divided by GCD of values in the tuple.

For a fixed x, Let us take the sum of the product of elements of all N-tuples.

Consider example

```
2
4 6
```

2 5

Let's consider each value of x one by one. Let g_x denote the sum of the product of values in tuples, where all values of the tuple are divisible by x.

$g_1 = 4*2+4*3+4*4+4*5+5*2+5*3+5*4+5*5+6*2+6*3+6*4+6*5 = 1^2*(4+5+6)*(2+3+4+5) = 210$
$g_2 = 4*2+4*4+6*2+6*4 = (4+6)*(2+4) = 2^2*(2+3)*(1+2) = 60$
$g_3 = 6*3 = (6)*(3) = 3^2*2*1 = 18$
$g_4 = 4*4 = (4)*(4) = 4^2*1*1 = 16$
$g_5 = 5*5 = (5)*(5) = 5^2*1*1 = 25$
$g_6 = 0$

It is easy to see from above that we can write the sum of products as the product of sum of possible choices for each element of the tuple.

For example, for $x = 2$, we could write g_2 as $(4+6)*(2+4)$ where first element of tuple could take values 4 and 6 and second value of tuple can take values 2 and 4. We can also divide each term of the product by x and multiply it separately. So g_2 becomes $2^N*(2+3)*(1+2)$ which is same as $2^N*(sum(3)-sum(1))*(sum(2)-sum(0))$ where $sum(n) = n*(n+1)/2$ as the sum of the first n natural numbers.

Consider for a general x, and an interval $[L_i, R_i]$, let a-th multiple of x be the largest multiple of x such that $l_x \leq L_i - 1$ and b-th multiple of x be the largest multiple of x such that $r_x \leq R$. It's not hard to prove that Sum of possible choices of multiple of x would be $x*(sum(b)-sum(a))$. Also, it is easy to see that $a = \lfloor \frac{L_i-1}{x} \rfloor$ and $b = \lfloor \frac{R_i}{x} \rfloor$

So we have $g_x = x^N * \prod_{i=1}^{N} [sum(\frac{R_i}{x}) - sum(\frac{L_i-1}{x})]$. From this, we can apply inclusion-exclusion to calculate the sum of product of values of tuple over all tuples with GCD x, say f_x.

Considering the above example again, we get
$f_1 = g_1 - f_2 - f_3 - f_4 - f_5 - f_6 = 107$
$f_2 = g_2 - f_4 - f_6 = 44$
$f_3 = g_3 - f_6 = 18$
$f_4 = g_4 = 16$
$f_5 = g_5 = 25$
$f_6 = g_6 = 0$

From this, the expected value of A/B as $\sum_{i=1}^{MX} \frac{f_i}{i}$ divided by the total number of ways to choose a tuple.

This gives us 50 points since calculating g is taking time $O(N*MX)$, one final optimization is needed.

For each g_x, let's ignore x^N part, we'll handle later. For a fixed interval $[L, R]$, it can be proved that $sum(\frac{R_i}{x})$ takes only $2*\sqrt{MX}$ distinct values. So $[sum(\frac{R_i}{x}) - sum(\frac{L_i-1}{x})]$ can take at most $4*\sqrt{MX}$ distinct values over all values of x. Moreover, each distinct value appears for some continuous values of x, allowing us to apply range updates.

Specifically, $[sum(\frac{R_i}{x}) - sum(\frac{L_i - 1}{x})]$ takes a different value when $\frac{R_i}{x} \neq \frac{R_i}{x-1}$ or $\frac{L_i - 1}{x} \neq \frac{L_i - 1}{x-1}$. We can precompute all such values of x for each value from 1 to MX, keep in sorted order, in order to get ranges with same value of $[sum(\frac{R_i}{x}) - sum(\frac{L_i - 1}{x})]$. We can use concept of difference arrays to apply these updates in $O(1)$ and then recovering the product array.

Lastly, need to take special care of 0s. Consider $x = 4$ and intervals $[4, 9]$ and $[9, 11]$. We can use the difference array to find which values of x have non-zero values for each interval.

Refer to the implementations below if anything is unclear.

Optimizations

- Try to reduce usage of modular operations. That would be enough.
- For applying updates, we'd need to compute modular inverses which take $O(log(MOD))$ time. If applied individually, Inverse would be calculated for nearly N*\sqrt{MX}$. We can group these inverses, reducing to computing inverses nearly MX times.

TIME COMPLEXITY

Precomputation takes $O(MX * \sqrt{MX})$ and each test case takes $O(N * \sqrt{MX} + MX * log(MOD))$

PREREQUISITES:

Combinatorics, Discrete Logarithm 55

PROBLEM:

The value of a set is the XOR sum of all elements in the set. The beauty of a sequence is defined as the sum of the values of all subsets. Let $F(N, B)$ be the number of non-negative integer sequences of length N with beauty B. Find the smallest B such that $F(N, B) \mod M = X$ or determine that none exist.

QUICK EXPLANATION:

Let p be the number of distinct bits (powers of 2) which appear in at least one element of the sequence. If we fix the p bits which appear in the sequence, there is only one possible beauty for the sequence, and there are $(2^N - 1)^p$ such sequences (this value $\mod M$ should be X). The minimum beauty is $2^{N-1} \cdot (2^p - 1)$ and it occurs when we choose the first p bits $(2^0, 2^1, \ldots, 2^{p-1})$. Finding the minimum value of p is a discrete logarithm problem.

EXPLANATION:

Let $a_1, a_2, \ldots, a_p$ be distinct bits which appear in at least one element of a sequence A.

Observation 1. The beauty of A is $2^{N-1}(2^{a_1} + 2^{a_2} + \cdots + 2^{a_p})$.

Proof

Like in most other problems involving bitwise operations, we should consider each bit by itself. Let's calculate the contribution of bit a_i to the answer.

The xor sum will only contain bit a_i if it appears an odd number of times. Thus, the contribution of bit a_i is $x2^{a_i}$, where x is the number of subsets of A which contain bit a_i an odd number of times. The value of x is well-known to be 2^{N-1}.

Proof

Consider the sequence B of length $N - 1$ which contains all elements of A except for an element with bit a_i, say x. There are 2^{N-1} ways to choose a subset from B. If that subset has bit a_i an odd number of times, then we cannot add x to the subset. If that subset has bit a_i an even number of times, we must add x to the subset. Thus, each subset of B corresponds to a subset of A with an odd number of occurrences.

We add the contribution for all bits to get $2^{N-1}(2^{a_1} + 2^{a_2} + \cdots + 2^{a_p})$.

Observation 2. There are $(2^N - 1)^p$ sequences with beauty $2^{N-1}(2^{a_1} + 2^{a_2} + \cdots + 2^{a_p})$.

Proof

Note that given the value of $2^{N-1}(2^{a_1} + 2^{a_2} + \cdots + 2^{a_p})$, we can uniquely determine the distinct bits $a_1, a_2, \ldots, a_p$ from the binary representation.

Consider bit a_1. We can choose any subset of the sequence to include bit a_1, except for the empty subset, since bit a_1 needs to appear at least once. There are $2^N - 1$ such subsets.

Since we have p bits, the total number of ways is $(2^N - 1)^p$.

The number of sequences only depend on the value of p. Thus, we should always choose the smallest p bits to calculate the beauty (since we want the minimum). The minimum beauty is for a certain p is $2^{N-1}\,(2^0 + 2^1 + \cdots + 2^{p-1}) = 2^{N-1}\,(2^p - 1)$.

A smaller value of p gives a smaller beauty, so we should find the minimum value of p such that $(2^N - 1)^p \equiv X \pmod M$. This problem is known as discrete logarithm and can be solved in $O(\sqrt{M})$ with baby-step giant-step. After we find p, we just need to calculate the minimum beauty and output it.

Note on calculating $2^N \mod M$ for N given as a string with a lot of digits:

Explanation

$2^{1234} \mod M = (2^{123} \mod M)^{10} \mod M * 2^4 \mod M$, so we can calculate $2^z \mod M$ for each prefix z of N without using any big integer classes.

There are also a few edge cases to take care of!

HIDDEN TRAPS:

- If $M = 1$, $B = 0$.
- If $X = 0$, we don't need a valid B (in other words, $B = 1$ unless $N = 1$).

PREREQUISITES:
Fast Fourier transform, basics of calculus, polynomials
PROBLEM:
You have to calculate $\sum_{i=0}^{N} \binom{p^i}{r}$ for given N, p and r modulo $m = 998\,244\,353$, where $r \le 10^6$.
QUICK EXPLANATION:
Consider $P(x) = \binom{x}{r}$ as a polynomial of x, then you may find $\sum_{i=0}^{N} P(p^i)$ as:

$$P(p^i) = \sum_{i=0}^{N} \sum_{k=0}^{n} a_k (p^i)^k = \sum_{k=0}^{n} a_k \sum_{i=0}^{N} p^{ki}.$$

EXPLANATION:
Note that $P_r(x) = \binom{x}{r}$ is the polynomial of x of degree r, which coefficients may be calculated in $O(r \log r)$. Indeed, for odd values $P_r(x)$ are recalculated from $P_{r-1}(x)$ and for even values of r we may rewrite it as follows:

$$P_{2r}(x) = \frac{x(x-1)\dots(x-2r+1)}{(2r)!} = P_r(x) \cdot P_r(x-r)$$

And transition from $P(x)$ to $P(x-r)$ may be done in $O(n \log n)$ time as well:

$$P(x-r) = \sum_{k=0}^{n} a_k (x-r)^k = \sum_{k=0}^{n} \sum_{i=0}^{k} a_k \binom{k}{i} x^i (-r)^{k-i} =$$

$$= \sum_{i=0}^{n} \frac{x^i}{i!} \sum_{k=i}^{n} (a_k k!) \cdot \frac{(-r)^{k-i}}{(k-i)!} = \sum_{i=0}^{n} \frac{c_i x^i}{i!}$$

Where c_i are coefficients defined by some particular convolution:

$$c_i = \sum_{k=i}^{n} (a_k k!) \cdot \frac{(-r)^{k-i}}{(k-i)!} = \sum_{k=0}^{n-i} a_{k+i} (k+i)! \cdot \frac{(-r)^k}{k!} = \sum_{k=0}^{n-i} \alpha_i \cdot \beta_{n-i-k}$$

Where $\alpha_k = \frac{(-r)^k}{k!}$ and $\beta_k = a_{n-k}(n-k)!$, thus c_i is a simple convolution of α and β. This convolution may be calculated as coefficients in the product of polynomials $A(x)$ and $B(x)$ having α_k and β_k as their coefficients correspondingly.

That means that the whole time needed to calculate $P_r(x)$ is:

$$T(n) = T(n/2) + O(n \log n) = O(n \log n)$$

Now that we know explicit coefficients of $P(x) = \binom{x}{r}$, our task is to calculate sum of $P(p^i)$ for i from 0 to N for given polynomial $P(x)$. Let it be that $P(x) = a_0 + a_1 x + \dots + a_n x^n$, then:

$$\sum_{i=0}^{N} P(p^i) = \sum_{k=0}^{r} a_k \sum_{i=0}^{N} p^{ik} = \sum_{k=0}^{r} a_k \frac{1 - p^{k(N+1)}}{1 - p^k}$$

Calculating this sum will take additional $O(r \log N)$ of time, which makes total complexity of the solution to be equal to $O(r \log rN)$.

PREREQUISITES:

Combinatorics

PROBLEM:

There are N points, the i-th point is at (i, A_i). All A_i are uniformly randomly generated from 1 to M. For all $0 \leq i \leq N-2$, segment i connects points i and $i+1$. You are given Q queries (l, r), meaning that you need to calculate the maximum height ($|A_i - A_{i+1}|$) of a segment in $[l, r)$. You can ask at most K questions of the form $(x1, x2, y)$ and you will be given the maximum height of a segment in $[x1, x2)$ which intersects with the horizontal line at y.

QUICK EXPLANATION 1:

Output $M-1$ for all queries.

EXPLANATION 1:

For subtask 1, $K = 3$ while $Q = 10$, so it's impossible to even ask 1 question per query! This prompts us to think whether we even need to ask any questions.

The constraint $R - L \geq 1000$ is really weird. It tells us that there are a lot of segments in the query. Generally, constraints in competitive programming are upper bounds, not lower bounds.

Also, $M \leq 10$ is very small. It kind of makes sense that there should be at least one segment with height $M-1$ (which is the maximum height) within those 1000 segments.

This is a long challenge, we can submit solutions without any sort of proof, so why not submit a solution which outputs $M-1$ for all queries? Turns out, it gets AC!

Let's analyze the probability that the answer is not $M-1$ for some query and let's call it x. Let's first calculate the probability that the height of some segment is not $M-1$. There are M^2 possibilities for choosing the first and second points for the segment. The only two possibilties when the segment can have a height of $M-1$ are $(1, M)$ and $(M, 1)$. The probability that we want is $y = 1 - \frac{2}{M^2}$.

Note that we can't just say that $x = y^{R-L}$ because the segments are not independent of each other. However, every other segment is independent, so we can ignore the segments between every other segment and say that $x < y^{\frac{R-L}{2}}$. If we calculate $y^{\frac{R-L}{2}}$, we find that it is around $4 \cdot 10^{-5}$ for $M = 10$.

QUICK EXPLANATION 2:

For each query (L, R), we will ask the question $(L, R, \frac{M}{2})$ and the answer of the question will be the answer of the query.

EXPLANATION 2:

For subtask 2, $K = 10$ while $Q = 10$, so we can ask 1 question per query.

The constraint $R - L \geq 100$ still tells us that there are a lot of segments in the query, so some randomization will probably be required again.

$M = 10^9$, so unlike the previous subtask, we can't just say that the answer for each query is $M-1$.

Our strategy will be to hit as many segments as possible with 1 question, and we will hope that the maximum height we get from this question will be the maximum height of the segments in the range of the query.

To hit as many segments as we can in the range, we should have $x1 = L$ and $x2 = R$. What about y?

If we choose $y = 1$ (which is the minimum possible value of A_i), then the line is unlikely to intersect with any of the segments. So, it makes sense that as we move the line towards the middle of 1 and M, the line will be able to intersect with more segments.

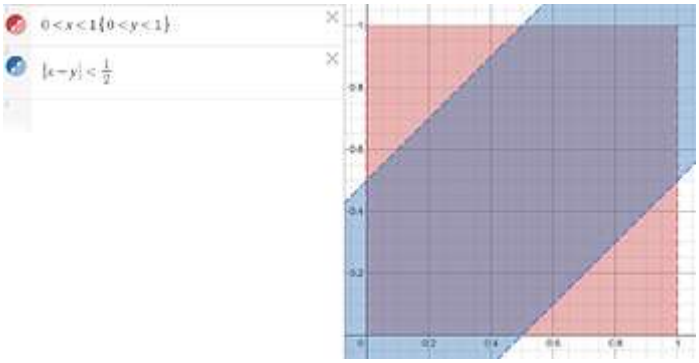
Our solution for this subtask is to, for each query (L, R), ask the question $(L, R, \frac{M}{2})$. We can submit this simple solution and find out that it gets AC!

We want to prove that the line $y = \frac{M}{2}$ will intersect the segment with the greatest height with a high probability. Notice that any segment with height greater than $\frac{M}{2}$ will intersect the line $y = \frac{M}{2}$. Let's instead prove that the line with the greatest height will always have a height greater than $\frac{M}{2}$.

Again, if the probability of failure is x, then we know that $x < y^{\frac{R-L}{2}}$, where y is the probability that a segment does not have a height greater than $\frac{M}{2}$.

We can define y more precisely: y is the probability that when we choose real numbers $0 < a, b < M$ (M is big so we can just treat the numbers as reals), $|a - b| < \frac{M}{2}$. We can scale down by M, so we are choosing real numbers $0 < a, b < 1$ and we want the probability that $|a - b| < \frac{1}{2}$.

We can find y with geometric probability:



The red square represents all (a, b) with $0 < a, b < 1$. The blue region represents all (a, b) with $|a - b| < \frac{1}{2}$. y is the fraction of the area of the red square that is also covered by the blue region. We can calculate that fraction to be $\frac{3}{4}$ (the two triangles have a combined area of $\frac{1}{4}$).

Lastly, we plug in the numbers and calculate $(\frac{3}{4})^{50}$, which is around $6 \cdot 10^{-7}$.

PREREQUISITES: Permutations and Combinations 11 and Factorials 4

PROBLEM STATEMENT:

You are given numbers N and K. There is a set S = {1, 2, … , n}. An **ordered tuple is a subsequence of this set.** Note, that we can r**earrange the elements within a tuple** in S.

The following are considered as **2 different tuples:**

$\{ (1,2), (3) \}$ and $\{ (2,1), (3) \}$

The above implies, that **rearrangement of elements within one tuple, is counted as two different valid configurations.**

While, the following **are the same:**

$\{(1,2), (3) \}$ and $\{ (3), (1,2) \}$

The above implies, that the r**earrangement of tuples in one valid configuration doesn't count twice.**

Find the **different ways of partitioning** the set S into **ordered tuples such that no tuple has a size greater than K.**

QUICK EXPLANATION:

For a tuple of size r(1 to k, as the limit can't exceed k), we **first compute** nPr (n having the original meaning). This gives us the n**umber of permutations of the tuples.** Now as **we have to take one tuple from size r, we have $(n-r)$ positions to be picked, and r sizes of tuples in which we can choose from.** Choose the $(n-r)$ positions in the r sized tuples and note that tuples of the same size in a set, can't be used as in condition 2. Likewise, you need to divide and get the correct answer.

EXPLANATION:

In this section, some symbols denote particular things. They are:

1. r – Current size of the tuple we are considering.
2. n – The number given to us. Same meaning as in the question.
3. K – The other number given to us. Same meaning as in the question.
4. aCb – The number of ways in which we can choose b elements from a elements. The formula for this is $(a!/((a-b)! * (b!)))$

Where $a!$ means n factorial, which is $a * (a-1) * \ldots * 2 * 1$.

5. aPb – The number of ways in which we can choose b elements from a elements and rearrange it. (Rearranging a tuple counts as 2 different tuples). The formula for this is $(a!/(a-b)!)$

Note: a and b are replaced by natural numbers, below, and not denoted as a and b. If you didn't know, $0!$ is not 0, but 1.

After reading the question, we come to know that it is about **arrangements.** The first thing which comes to our mind when we hear "arrangements" is **permutations and combinations. With this as our prerequisite, and some easy observations, we will get our answers.**

Observation / Basic Idea:

1. One **mistake** that 99.9% of people make when it comes to this is **overcounting** (i.e counting the same sequence a repeated number of times). The idea for us to **encounter this is that when we choose to work**

on a tuple of size r, we will only consider tuples of size $<= r$

Proof that it will work

Suppose there is a combination which has one tuple of size 2, and one tuple of size 3. When we consider tuples of size 2, we will not take this combination into consideration as the sizes of all its tuples are not $<= 2$.

When we consider tuples of size 3, we will take this combination into consideration as the size of all its tuples is $<= 3$.

When we consider tuples of size 4, we will not consider this combination because there is no tuple of size 4. Likewise, all the other arrangements will be chosen only once.

–End of proof–

So, according to our basic idea, our aim is to **consider all tuples (from** 1 **to** k**), and when considering them, we make a set which has one tuple of that size, and all the other tuples are <= the size we are considering currently.**

For example, a valid configuration when we are currently considering a tuple of size 3 $(N = 6)$ is $(\{1, 2, 3\}, \{4, 5, 6\})$ or $(\{1, 2, 4\}, \{5, 6\}, \{3\})$ and many more…

Now, what will be our formula?

Firstly, we have to add all the number of arrangements which we have found out are made previously for the sizes $< r$.

Secondly, we have to choose a tuple of size r**. So there are** nCr **ways to choose it, and as we can rearrange it, we add** nPr**. (because** nPr **is** nCr **+ rearranging)**

Now, a **valid arrangement shall contain that tuple, and many other tuples of size** $<= r$. Suppose, for $N = 4, K = 2$, we have chosen a tuple of size 2. Now there are 2 elements left, and we need to choose 2 of them. As rearrangement is possible, we will **multiply** our answer by $2P2$. On the other hand, we can't arrange the tuples in our set, thus we divide it by 2.

Note: We **multiply and divide because for each valid tuple the configuration is possible, and not only for a single one.** We **add the answers** to all the calculations as has to be counted only once.

Now suppose $N = 6, K = 2, r = 2$. We choose a tuple of size 2. Now, we have 4 elements in which we can choose 2, thus we multiply our answer by $4P2$. Now, we have 2 elements and we have to choose 2, thus we multiply our answer by $2P2$. We can arrange the tuples of size 2 in 6 ways. Thus we divide our answer by 6, etc.

Similarly, it is **not that we have to only choose a tuple of size** r **in our set. We can make a tuple of size** 2**, when** r **is** 3. So our answer is, choose X tuples of size Ai. Ai denotes the **size** of the ith tuple. Here the **summation of all** Ai **is equal to** N.

Now, iterate through A which we have made, and write down the number of tuples of size B (for all B from 1 to N). We **divide the answer we got by this arrangement by the factorial of the numbers written.**

Example:

$N = 8, K = 3.$

One possible way is to choose 3 tuples of size 2, 2 tuples of size 1.

Thus whatever our answer will be, we will **divide it by** $((3!) + (2!))$

This is because **rearrangement of tuples will not count as two valid configurations,** thus, there are $3!$ ways to arrange the tuples of size 2, and $2!$ ways to arrange the tuples of size 1. Thus, we divide our answer by this.

SOLVING SUBTASKS:

Subtask A

$N = 4, K = 3$

1. $r = 1$. The number of permutations which can be made by only tuples of size 1 is 1.

Thus, $answer = 1$.

2. $r = 2$. According to the **formula we have taken out above**, the answer will be:

$Previouscalculations + 4P2 + ((4P2 * 2P2)/2!)$

$=> 1 + (24/2) + (((24/2) * (2/1))/2)$

$=> 1 + 12 + (12 * 2)/2$

$=> 1 + 12 + 12$

$=> 25$

3. $r = 3$.

Answer will be:

$Previouscalculations + 4P3$

$=> 25 + (24/1)$

$=> 25 + 24$

$=> 49$

So our **final answer is** 49**.**

Subtask B

$N = 5, K = 3.$

1. $r = 1$. The number of permutations which can be made by only tuples of size 1 is 1.

Thus, $answer = 1$.

2. $r = 2$. According to the **formula we have taken out above,** the answer will be:

$Previouscalculations + 5P2 + ((5P2 * 3P2)/(2!))$

$=> 1 + (120/6) + (((120/6) * (6/1))/(2!))$

$=> 1 + 20 + (20 * 6/2)$

$=> 1 + 20 + 60$

$=> 81$

3. $r = 3$.

Answer will be:

$Previouscalculations + 5P3 + (5P3 * 2P2)$

$=> 81 + (120/2) + ((120/2) * (2/1))$

$=> 81 + 60 + (60 * 2)$

$=> 81 + 60 + 120$

$=> 261$

Thus, **our final answer is** 261**.**

Note: Here we **did not divide by** $2!$**, as the sizes of tuples were different, i.e one tuple was of size** 2**, and one of size** 3**.**

Subtask C

$N = 6, K = 3$.

1. $r = 1$. The number of permutations which can be made by only tuples of size 1 is 1.

Thus, $answer = 1$.

2. $r = 2$. According to the formula we have taken out above, the answer will be:

$Previouscalculations + 6P2 + ((6P2 * 4P2)/(2!)) + ((6P2 * 4P2 * 2P2)/(3!))$

$=> 1 + (720/24) + ((720/24) * (24/2))/(2!)) + (((720/24) * (24/2) * (2/1))/(6))$

$=> 1 + 30 + ((30 * 12)/2) + ((30 * 12 * 2)/6)$

$=> 31 + 180 + 120$

$=> 331$

3. $r = 3$.

Answer will be:

$Previouscalculations + 6P3 + ((6P3 * 3P3)/(2!)) + (6P3 * 3P2)$

$=> 331 + 120 + ((120 * 6)/2) + (120 * 6)$

$=> 451 + 360 + 720$

$=> 1531$

So, **our final answer is 1531.**

EXTRAS:

Find the answer when $N = 5, K = 3$ and we are not allowed to rearrange the elements in set S.

Thus, a tuple {1, 3} will **not be valid here.**

I hope I could make you understand the concept of this problem.

PREREQUISITES:

Graph theory, Combinatorics

PROBLEM:

A short and concise description of the problem statement.

QUICK EXPLANATION:

Given a graph with N nodes and M - 1 edges. You have also been given K colours and you are required to find the number of ways to colour the nodes which can be reached from-and-to each other using the same colour, and the ones not reachable in both ways, with different colours.

EXPLANATION:

Basically, it's clear from the problem itself that the nodes which are reachable from-and-to each other form strongly connected components.
So, we just need to find the number of strongly connected components. Let this count be 's'. Then if K is less than 's', answer is -1. Else the answer is k! / (k - s)!.

PREREQUISITES:

Combinatorics, dynamic programming

PROBLEM:

How many ways are there to color a tree of n nodes with k colors so that for every two nodes with the same color, all nodes in the path between them also have the same color?

QUICK EXPLANATION:

The lowest common ancestor of each color determines the coloring completely. In other words, once we select a subset of the nodes S and color them all differently, the color of every node a can be obtained by walking the path from a to node 1 and stopping at the first node in S. The color of a is the color of that final node.

Thus, the answer is the number of possible choices for S.

Note that the root must be the LCA of one of the colors. If there are i colors, then there are $\binom{n-1}{i-1}$ ways to choose the non-root LCAs of the other colors, $\binom{k}{i}$ ways to choose the colors, and $i!$ ways to assign these colors to the LCAs. Thus, there are $\binom{n-1}{i-1}\binom{k}{i}i!$ ways all in all. Summing across all valid i, we get:

$$\sum_{i=1}^{k} \binom{n-1}{i-1}\binom{k}{i}i!$$

Note that the answer is independent of the tree structure! It only depends on n and k.

To compute this, we precompute $\binom{n}{r}$ and $n!$ modulo 10^9+7 for all $0 \le n, r \le 50$ using Pascal's identity, and then compute the sum above (modulo 10^9+7) in $O(k)$ time.

EXPLANATION:

Valid colorings

Observe that the constraint "for every two nodes with the same color, all nodes in the path between them also have the same color" is equivalent to the following: **For every color, if you remove all nodes not colored with that color, along with the edges adjacent to them, then the remaining subgraph is connected.** This means that every color induces a **subtree**!

We can visualize this in another way by rooting the tree at some node, say, node 1. In this case, if you view only the nodes that are of some color, then it also looks like a rooted tree, rooted at some node. This "root" is actually the **lowest common ancestor** (LCA) of all the nodes of that color.

What's amazing is that, once we fix the lowest common ancestor for every color, the color of every other node is now completely determined! To see this, suppose we fix the nodes $S = \{i_1, i_2, \ldots, i_m\}$ as the LCAs of the colors that appear in the tree. Then for every node i, its color must be the color of its lowest ancestor that is in S! This can be seen intuitively (e.g. to draw some examples to visualize this), but here's a proof:

Suppose i_k is the lowest ancestor of i that is in S, and suppose $\text{color}(i_j) = \text{color}(i)$ for some $i_j \in S$. Now, suppose for the purpose of contradiction that $j \neq k$. Then there are two cases, depending on whether i_j is a descendant of i_k or not:

- Suppose i_j is a descendant of i_k. Then i_j cannot be an ancestor of i, because i_k is the lowest ancestor of i in S. This means that $LCA(i_j, i) \neq i_j$, which contradicts the fact that i_j is the LCA of all nodes of the same color as i_j.
- Suppose i_j is not a descendant of i_k. Note that i is a descendant of i_k, which means that i_k is on the path from i to i_j, which fails our constraint.

In either case, we have a contradiction. Thus, $j = k$ after all.

This means that the number of valid colorings depend on the number of choices for the set S of LCAs, and also on the colors that we give these nodes. But notice that we can choose the LCAs without actually looking at the edges! It means that *the answer is independent on the edges, and is only dependent on n and k!

Counting valid colorings

Now, we know what we need to count. We need to count the number of valid selections of the set S, and the number of assignments of colors among them. S can be any set of nodes, with one condition: node 1 must be in S. (Why?)

Now, let's compute the answer. Suppose exactly i colors appear in the tree. How many choices for S are there? Well, S has i elements, but one element has already been chosen for us, namely node 1, so we need to select the remaining $i-1$ nodes among the remaining $n-1$ nodes. There are $\binom{n-1}{i-1}$ choices. Next, we need to assign colors to these nodes. First, we must select i colors among the k available. There are $\binom{k}{i}$ choices. Finally, how many ways are there to assign these colors to the nodes? There are i choices of colors for the first node, $i-1$ for the second node, $i-2$ for the third node, etc. Therefore, there are $i(i-1)(i-2)\cdots 2 \cdot 1 = i!$ assignments. Therefore, the number of colorings of a tree with exactly i colors is $\binom{n-1}{i-1}\binom{k}{i}i!$. Summing across all valid values of i, we get the following answer:

$$\sum_{i=1}^{\min(n,k)} \binom{n-1}{i-1}\binom{k}{i}i!$$

Computing this value modulo 10^9+7 can be done in $O(k)$ time assuming we have already computed the binomial coefficients and factorials modulo 10^9+7. This can be done with dynamic programming:

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

The base cases are $0! = 1$ and $\binom{n}{0} = \binom{n}{n} = 1$.

Since n and r can only reach up to 50 in this problem, we can quickly tabulate all the needed values.

Time Complexity:

$O(n+k)$

Prerequisites : combinatorics , stars and bars 6 , dp

Problem statement :

There are K boards to be hung on ‘N’ poles . Each board can be of length 1 or 2 and no two boards can share a common pole. Find the number of ways of doing so.

METHOD 1 : COMBINATORICS →

Subtask A :

N = 8 , K = 2

Let’s assume that the lengths of the two boards are L1 and L2

The first board is after X1 poles

The number of poles between both the boards is Y

There are X2 poles after the second pole

How many places are there to fill ?

A board takes the gap between two poles and not the poles , therefore we need to count the number of gaps between two consecutive poles and not the number of poles

The number of gaps between consecutive poles where the N is the number of poles is N-1

Therefore we have N-1 places to fill the boards and not N

Here is how it would look :

X1 L1 Y L2 X2 = 8-1 = 7

X1 + L1 + Y + L2 + X2 = 7

where

X1 , X2 >= 0

Y >= 1 : Because any pole cant have 2 boards

So now there is a problem as Y should be greater than 1 , so to make it 0 , we subtract 1 from both the sides and let our new variable Y’ be Y-1 and our new equation would be :

X1 + L1 + Y’ + L2 + X2 = 7-1 = 6

Now here , X1 , Y’ , X2 >= 0 , therefore we can apply the formula

We know that L1 , L2 should be 1 or 2

So let us take those cases and find out answers for each of them and we will add the answers for all the cases cases and we would get our answer 🙂

Case 1 : L1 = 1 , L2 = 1 :

X1 + L1 + Y’ + L2 + X2 = 7-1 = 6

X1 + 1 + Y’ + 1 + X2 = 6

X1 + Y’ + X2 = 4

Formula for this is (3+4-1)C(3-1) = 6C2 = 15

Case 2 : L1 = 2 , L2 = 1 :

X1 + L1 + Y' + L2 + X2 = 7-1 = 6

X1 + 2 + Y' + 1 + X2 = 6

X1 + Y' + X2 = 3

Formula for this is (3+3-1)C(3-1) = 5C2 = 10

Case 3 : L1 = 1, L2 = 2 :

X1 + L1 + Y' + L2 + X2 = 7-1 = 6

X1 + 1 + Y' + 2 + X2 = 6

X1 + Y' + X2 = 3

Formula for this is (3+3-1)C(3-1) = 5C2 = 10

Case 4 : L1 = 2 , L2 = 2 :

X1 + L1 + Y' + L2 + X2 = 7-1 = 6

X1 + 2 + Y' + 2 + X2 = 6

X1 + Y' + X2 = 2

Formula for this is (3+2-1)C(3-1) = 4C2 = 6

15 + 20 + 6 = 41

OBSERVATION : Here we observe that the answer for both case 2 and case 3 is the same because only the values of L1 and L2 are changing and it doesn't matter. Instead of doing case 3 we could have just done 2* 5C2 = 20 = Case 2 + Case 3

Answer : 41

Subtask B :

N = 9 , K = 3

Number of places to fill - 8

Number of places before board 1 - X1

Number of places between board 1 and board 2 - Y1

Number of places between board 2 and board 3 - Y2

Number of places after board 3 - X2

Lengths of boards - L1 , L2 , L3

X1 L1 Y1 L2 Y2 L3 X2

Equation : X1 + L1 + Y1 + L2 + Y2 + L3 + X2 = 8

Y1 , Y2 >= 1 , subtracting 1 both sides two times (one time for Y1 and another time for Y2)

New equation : X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 8-1-1 = 6

Case 1 : L1 = L2 = L3 = 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 6

X1 + 1 + Y'1 + 1 + Y'2 + 1 + X2 = 6

X1 + Y'1 + Y'2 + X2 = 3

Number of ways to do this : (3+4-1)C(4-1) = 6C3 = 20

Case 2 : L1 or L2 or L3 = 2 , other two 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 6

X1 + 2 + Y'1 + 1 + Y'2 + 1 + X2 = 6

X1 + Y'1 + Y'2 + X2 = 2

Number of ways we can choose L1 or L2 or L3 = 2 → 3C1 = 3

Number of ways to do the above equation : (2+4-1)C(4-1) = 5C3 = 10

Total number of ways → 3*10 = 30

Case 3 : Any two of L1 , L2 , L3 = 2 , the other one is 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 6

X1 + 2 + Y'1 + 2 + Y'2 + 1 + X2 = 6

X1 + Y'1 + Y'2 + X2 = 1

Number of ways we can choose two boards to have length 2 → 3C2 = 3

Number of ways to do the above equation : (1+4-1)C(4-1) = 4C3 = 4

Total number of ways → 3*4 = 12

Case 4 : All of them L1 , L2 , L3 = 2 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 6

X1 + 2 + Y'1 + 2 + Y'2 + 2 + X2 = 6

X1 + Y'1 + Y'2 + X2 = 0

Number of ways to do the above equation : (0+4-1)C(4-1) = 3C3 = 1

20 + 30 + 12 + 1 = 63

Answer : 63

Now let us do Subtask D as K = 3 →

Subtask D :

N = 11 , K = 3

Number of places to fill - 10

Number of places before board 1 - X1

Number of places between board 1 and board 2 - Y1

Number of places between board 2 and board 3 - Y2

Number of places after board 3 - X2

Lengths of boards - L1 , L2 , L3

X1 L1 Y1 L2 Y2 L3 X2

Equation : X1 + L1 + Y1 + L2 + Y2 + L3 + X2 = 10

Y1 , Y2 >= 1 , subtracting 1 both sides two times (one time for Y1 and another time for Y2)

New equation : X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 10-1-1 = 8

Case 1 : L1 = L2 = L3 = 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 8

X1 + 1 + Y'1 + 1 + Y'2 + 1 + X2 = 8

X1 + Y'1 + Y'2 + X2 = 5

Number of ways to do this : (5+4-1)C(4-1) = 8C3 = 56

Case 2 : L1 or L2 or L3 = 2 , other two 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 8

X1 + 2 + Y'1 + 1 + Y'2 + 1 + X2 = 8

X1 + Y'1 + Y'2 + X2 = 4

Number of ways we can choose L1 or L2 or L3 = 2 → 3C1 = 3

Number of ways to do the above equation : (4+4-1)C(4-1) = 7C3 = 35

Total number of ways → 3*35 = 105

Case 3 : Any two of L1 , L2 , L3 = 2 , the other one is 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 8

X1 + 2 + Y'1 + 2 + Y'2 + 1 + X2 = 8

X1 + Y'1 + Y'2 + X2 = 3

Number of ways we can choose two boards to have length 2 → 3C2 = 3

Number of ways to do the above equation : (3+4-1)C(4-1) = 6C3 = 20

Total number of ways → 3*20 = 60

Case 4 : All of them L1 , L2 , L3 = 2 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + X2 = 8

X1 + 2 + Y'1 + 2 + Y'2 + 2 + X2 = 8

X1 + Y'1 + Y'2 + X2 = 2

Number of ways to do the above equation : (2+4-1)C(4-1) = 5C3 = 10

56 + 105 + 60 + 10 = 231

Answer : 231

Subtask C :

N = 10 , K = 4

Number of places to fill - 9

Number of places before board 1 - X1

Number of places between board 1 and board 2 - Y1

Number of places between board 2 and board 3 - Y2

Number of places between board 3 and board 4 - Y3

Number of places after board 4 - X2

Lengths of boards - L1 , L2 , L3, L4

X1 L1 Y1 L2 Y2 L3 Y3 L4 X2

Equation : X1 + L1 + Y1 + L2 + Y2 + L3 + Y3 + L4 + X2 = 9

Y1 , Y2 , Y3 >= 1 , subtracting 1 both sides three times (one time for Y1 and another time for Y2 and another time for Y3)

New equation : X1 + L1 + Y'1 + L2 + Y'2 + L3 + Y'3 + L4 + X2 = 9-1-1-1 = 6

Case 1 : L1 = L2 = L3 = L4 = 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + Y'3 + L4 + X2 = 6

X1 + 1 + Y'1 + 1 + Y'2 + 1 + Y'3 + 1 + X2 = 6

X1 + Y'1 + Y'2 + Y'3 + X2 = 2

Number of ways to do this : (2+5-1)C(5-1) = 6C4 = 15

Case 2 : L1 or L2 or L3 or L4 = 2 , other three 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + Y'3 + L4 + X2 = 6

X1 + 2 + Y'1 + 1 + Y'2 + 1 + Y'3 + 1 + X2 = 6

X1 + Y'1 + Y'2 + Y'3 +X2 = 1

Number of ways we can choose L1 or L2 or L3 or L4 = 2 → 4C1 = 4

Number of ways to do the above equation : (1+5-1)C(5-1) = 5C4 = 5

Total number of ways → 4*5 = 20

Case 3 : Any two of L1 , L2 , L3, L4 = 2 , the other two is 1 :

X1 + L1 + Y'1 + L2 + Y'2 + L3 + Y'3 + L4 + X2 = 6

X1 + 2 + Y'1 + 2 + Y'2 + 1 + Y'3 + 1 + X2 = 6

X1 + Y'1 + Y'2 + Y'3 + X2 = 0

Number of ways we can choose two boards to have length 2 → 4C2 = 6

Number of ways to do the above equation : (0+5-1)C(5-1) = 4C4 = 1

Total number of ways → 6*1 = 6

Now , if we continue like this in the next step , our RHS would become negative 😕 which we can do in 0 ways , so we can skip this and directly find our answer

15 + 20 + 6 = 41

Answer : 41

METHOD 2 : DYNAMIC PROGRAMMING (DP) →

Let us define our dp table as follows :

Dp[i][j] = number of ways to arrange i signs between j poles

And our answer would be stored in dp[K][N]

Base case :

As we see , to hang i boards , we need at least 2i places

Therefore dp[i][2i] = 1 and dp[i][x] = 0 where x < 2i

Dp[1][3] = (j - 1) + (j - 2) = 2 + 1 = 3

Dp[1][j] = (j-1) + (j-2) = 2*j - 3 →

Because there is only one board and there are j places so we can put it in j-1 ways and if the length is 2 then we can put it in j-2 places .

Now is the recursion :

CASE 1 : Either we consider the jth pole :

Case 1 : The current length of the board we are considering right now is 1 :

We can do it just by adding the current board to the end , So we have i-1 boards (-1 because of excluding current board) and in j-2 places (-2 because the length of the board is 1)

Therefore the answer to this is dp[i-1][j - 2]

Case 2 : The current length of the board we are considering right now is 2 :

We can do it just by adding the current board to the end , So we have i-1 boards (-1 because of excluding current board) and in j-3 places (-3 because the length of the board is 2)

Therefore the answer to this is dp[i-1][j - 3]

CASE 2 : If we don’t consider the jth pole :

Then the answer is simply dp[i][j-1]

Therefore , dp[i][j] = dp[i][j-1] + dp[i-1][j-2] + dp[i-1][j-3]

Now , let us start filling the dp table :

Here we also note that the same dp table will be the same for all the testcases , so we can only maintain only one dp table . The maximum value of k is 4 and that of N is 11 . So we will do dp table of size 11*4

Rows are K and the columns are N

Here is how the dp table would look after doing the base case :

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	0	1									

	1	2	3	4	5	6	7	8	9	10	11
2	0	0	0	1							
3	0	0	0	0	0	1					
4	0	0	0	0	0	0	0	1			

Dp[1][3] = 2*3 - 3 = 3

Dp[1][4] = 2*4 - 3 = 5

Dp[1][5] = 2*5 - 3 = 7

Dp[1][6] = 2*6 - 3 = 9

Dp[1][7] = 2*7 - 3 = 11

Dp[1][8] = 2*8 - 3 = 13

Dp[1][9] = 2*9 - 3 = 15

Dp[1][10] = 2*10 - 3 = 17

Dp[1][11] = 2*11 - 3 = 19

And our dp table will be :

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	3	5	7	9	11	13	15	17	19
2	0	0	0	1							
3	0	0	0	0	0	1					
4	0	0	0	0	0	0	0	1			

Now k = 2

Dp[2][5] = dp[2][4] + dp[1][2] + dp[1][3] = 1 + 1 + 3 = 5

Dp[2][6] = dp[2][5] + dp[1][3] + dp[1][4] = 5 + 3 + 5 = 13

Dp[2][7] = dp[2][6] + dp[1][4] + dp[1][5] = 13 + 5 + 7 = 25

Dp[2][8] = dp[2][7] + dp[1][5] + dp[1][6] = 25 + 7 + 9 = 41 (SUBTASK A)

Dp[2][9] = dp[2][8] + dp[1][6] + dp[1][7] = 41 + 9 + 11 = 61

Dp[2][10] = dp[2][9] + dp[1][7] + dp[1][8] = 61 + 11 + 13 = 85

Dp[2][11] = dp[2][10] + dp[1][8] + dp[1][9] = 85 + 13 + 15 = 113

Now the dp table would be like this :

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	3	5	7	9	11	13	15	17	19
2	0	0	0	1	5	13	25	41	61	85	113
3	0	0	0	0	0	1					

	1	2	3	4	5	6	7	8	9	10	11
4	0	0	0	0	0	0	0	1			

Now k = 3 :

Dp[3][7] = dp[3][6] + dp[2][4] + dp[2][5] = 1 + 1 + 5 = 7

Dp[3][8] = dp[3][7] + dp[2][5] + dp[2][6] = 7 + 5 + 13 = 25

Dp[3][9] = dp[3][8] + dp[2][6] + dp[2][7] = 25 + 13 + 25 = 63 (SUBTASK B)

Dp[3][10] = dp[3][9] + dp[2][7] + dp[2][9] = 63 + 25 + 41 = 129

Dp[3][11] = dp[3][10] + dp[2][8] + dp[2][10] = 129 + 41 + 65 = 231 (SUBTASK D)

Now our dp table would be like this :

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	3	5	7	9	11	13	15	17	19
2	0	0	0	1	5	13	25	41	61	85	113
3	0	0	0	0	0	1	7	25	63	129	231
4	0	0	0	0	0	0	0	1			

Now k = 4 :

Dp[4][9] = dp[4][8] + dp[3][6] + dp[3][7] = 1 + 1 + 7 = 9

Dp[4][10] = dp[4][9] + dp[3][7] + dp[3][8] = 9 + 7 + 25 = 41 (SUBTASK C)

Dp[4][11] = dp[4][10] + dp[3][8] + dp[3][9] = 41 + 25 + 63 = 129

Now our dp table would look like this :

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	3	5	7	9	11	13	15	17	19
2	0	0	0	1	5	13	25	41	61	85	113
3	0	0	0	0	0	1	7	25	63	129	231
4	0	0	0	0	0	0	0	1	9	41	129

Answers :

Subtask A : 41

Subtask B : 63

Subtask C : 41

Subtask D : 231

Hope you understood both the methods 🙂

Special thanks to my mentor of ICPC for schools , @sarthakmanna sir and @avijit_agarwal sir for teaching me stars and bars method !

PRE-REQUISITES:

Concept of Modular Exponentiation [1] and Fermat's Little Theorem [1], Applications of FFT (All Possible Sums) [7], Basic Combinatorics [2] (The link might be more than what you need), General concepts of Modular set - like Generator of the Modular Set [2] and Primitive Root [5] (It just sound scary - its a trivial thing).

PROBLEM:

Given an array A with N elements, we make a brute force generator to generate sequences of length K to crack password. Our brute force generator generates all N^K sequences. The password is product of the K chosen elements modulo P. We need to find how many times each integer from 0 to $P-1$ is tried by the sequences generated by our generator modulo 998244353.

QUICK-EXPLANATION:

Key to AC- All possible sum (FFT) can be used if we change multiplication to addition!

We need to handle the case of 0 separately. Lets use a frequency array F to store frequency of elements. Let $F[0]$ denote the frequency of 0. The solution for case of 0 would be $N^K - (N - F[0])^K$.

Now we
G be th

Thus, t
... $* A_{i}$

Given t
the pro
many ways can we chose K of them to get a (modular) sum of r, for each r from 0 to $P-2$.

Quote Share

To solve above, we go by method of All Possible Sums. We make a polynomial $P(x)$ such that:

- The zero'eth coefficient (we are using this term for convenience) corresponds to power of x^0. Similarly first coefficient corresponds to power x^1. Hence, $i^{'}th$ coefficient corresponds to power x^i.
- The value of $i^{'}th$ coefficient is nothing but frequency of i in array A (after transformation).

Now, since we can chose K elements, our answer lies in the $K^{'}th$ power of this polynomial. To find it, we do the following:

- K is large, so we cannot do normal multiplication. Instead, we will use the same algorithm of Fast Exponentiation to calculate it - just this time on polynomials instead of numbers! In other words, we calculate $P(x)$, $P(x)^2$, $P(x)^4$... and so on and use them to find $P(x)^K$ in $O(logK)$ time factor.
- The terms in the polynomial will be too many in higher powers! But hey, we are interested in $modular$ sum, not normal sum. What I mean is, take any power, say x^i where $i \geq P-1$. Note that the coefficient of x^i represents in how many ways we can get i by summing. But we are only interested in values from 1 to $P-1$ (we calculated for 0 already), and hence i's from 0 to $P-2$. So instead of keeping a separate term, and then manipulating result at end, we add coefficient of x^i to $x^{i\%(P-1)}$ and discard x^i. Hence number of terms are always $\leq P-1$.

The only thing that's left then trivial manipulation to see answer for which y_i corresponds to which A_i and print the result.

EXPLANATION:

The editorial will have following sections-

- Transforming the problem from Product to Sum.
- Bringing in Polynomial - Their interpretation, how they can help and the challenges.
- Fast Exponentiation on above polynomial to get the answer

Mostly, the detailed explanation is aimed towards not-the-top-star-coders because the top coders already got everything they needed in the Quick Explanation Section. In case any doubt remains, they are requested to just skim through relevant sections.

There are some concept-checks in editorial - there solution is also in the bonus corner.

Transforming the problem from Product to Sum

This was the first step to solve this problem. And there are some things we need to realize.

For any problem related to product or divisions, the case of 0 has to be - almost always - either handled separately or seen with care.

So going by that norm, we will handle it separately. Number of times 0 is used as a password can be easily calculated as we only need count of sequences with at least 1 zero. This can be easily found as-

Count Sequences with at least one zero = Total sequences - Count of Sequences with no zeroes at all.

Let $F[0]$ be the frequency of 0 in A. Hence, the answer for 0 becomes $N^k - (N - F[0])^k$. If you do not get how, its given in bonus corner.

Concept Check - Why are we not using the nCk formula here?

Now, we need the solution for passwords from 1 to $P-1$.

Now, obviously the product of K terms is a difficult thing to handle. Not only it overflows the standard data type range - even using Big Integer etc is going to be very costly, especially for $K \approx 10^9$.

The most common trick we apply for these cases is to somehow change multiplication to addition. For different questions, its done in different ways, like taking logs etc. For this question, taking logs is not desirable as we are doing modular arithmetic, hence we use the other concept - Generator or Primitive Root. (refer to pre-requisites and bonus corner for more)

Using the methods given in pre-requisite links, we can find the Primitive Root/Generator for the set, i.e. G. Note that when I say set, I mean the set of elements from 1 to $P-1$. Now, every element in A can be represented as some power of G. In other words, we replace A_i with y_i such that we have $G^{y_i} = A_i$.

Now, the product of $A_{i1} * A_{i2} * ... * A_{ik} \% P$ becomes sum (of powers) of G. That is, $A_{i1} * A_{i2} * ... * A_{ik} \% P$ now becomes $G^{(y1+y2+...+yk)\%(P-1)} \% P$.

Concept check - Why are the powers modulo P-1 instead of P?

Now, our problem transforms from “Finding how many times product of K elements equals to i for all i from $[1, P-1]$” to "Finding in how many ways sum of $y_1 + y_2 + ... + y_k \% P$ yields c such that $G^c \% P \in [1, P-1]$ for all $c \in [0, P-2]$ (Note modulo of power is $P-1$ and each power from 0 to $P-2$ represents a distinct integer in range $[1, P-1]$)

Bringing in the Polynomial - Their interpretation and how they can help.

Our problem is now reduced to finding in how many ways we can chose K elements from our transformed A such that their modular sum is c for all $0 \le c \le P-2$. This can be solved by using All Possible Sums algorithm.

The original problem says that, given 2 arrays, find what are the possible sums and how many ways are possible to generate them. Seems pretty similar! The only changes are this:

- All possible sums (given in FFT pre-requisite link) does it for 2 arrays, and assumes we want all possible sums if we chose one element from each.
- It can be easily generalized. We know that all our elements are to be chosen from A. This is similar to having K copies of array A and having to choose one element from each. This is why you will see we go for $K^{'}th$ power of frequency polynomial.

To begin with solving the problem, we shall make a frequency polynomial. This is a polynomial where $i^{'}th$ coefficient (corresponding to power x^i) has value $F[i]$ where $F[i]$ is frequency of i in A. Obviously, our polynomial has $0-based$ indexing (zeroth coefficient corresponds to x^0). Note that this is done in the modified/transformed array A where A_i is replaced by y_i such that $G^{y_i} = A_i$. (Concept check - Why does this polynomial work? This also deals with how it helps and is at bonus corner)

As already justified above, our answer shall lie in the $K^{'}th$ power of this polynomial. But there are 2 challenges we face here:

- K is very large. Using normal FFT will take a LOT of time for $K \approx 10^9$.
- The number of terms is going to kill us as well. For $K \approx 10^9$, you can imagine how many terms the $K^{'}th$ power will have.

We will resolve these difficulties in the next section.

Overcoming Challenges using our old and friendly Fast Exponentiation Trick

We had 2 challenges in previous section. One pertaining to K being very high - hence it being very time consuming to calculate $K^{'}th$ power of the polynomial. And the other pertained to the $K^{'}th$ power having too many terms to cause memory and time limits to get exceeded.

You will see that both of these are actually handled by very basic fundamentals.

Firstly, lets deal with high number of terms first. It is a legit question, that how can we solve the problem with this method if the number of terms are $\approx 10^9$? The answer is simple - we are interested in $modular$ sum and not normal sum! All Possible Sums solves for normal sum, but it is faster if it has to be done modulo P. Actually, since the sum we are finding are powers of G, they are modulo is $P-1$, but you get the flow!

Now, what happens when terms increase? Terms increase because we got a new power of x, say x^i where $i \geq (P-1)$. Note that, we are interested in finding how many ways we can select $(y_1, y_2, ..., y_k)$ and get a value c where $0 \leq c \leq P-2$. So we know that ultimately the number of ways to get i (which is the coefficient of x^i) are to be added to number of ways to get $i\%(P-1)$ to get the final answer (because we are interested in $modular$ sum and not normal sum). So why not do that now?

All that I'm saying is, whenever we get a term x^i where $i \geq P-1$, we simply add its contribution to $x^{i\%(P-1)}$ and discard the term x^i. This keeps number of terms at any time $O(P)$ and the concept behind the polynomial stays intact!

Now all that we need to handle is finding the $K^{'}th$ power. This can be easily done using fast exponentiation trick - except this time on polynomials instead of numbers! Let $P(x)$ be our frequency polynomial. We will calculate $P(x)$, $P(x)^2$, $P(x)^4$, $P(x)^8$, ... and so on to finally obtain $P(x)^K$ using their combination. The multiplication happens as described earlier (to keep number of terms in check).

So if you've understood till here, pat yourself on the back because you are now well equipped to solve this problem. You will find referring to setter's code very much easy - just know that he has used some algebra library (which you should as well) instead of writing the code from scratch. So just trust those function names and read the code with a pinch of abstraction.

SOLUTION

Setter

```
// In The Name Of The Queen
#include<bits/stdc++.h>
using namespace std;
const int LG = 18, N = 1 << LG, Mod = 998244353, LGK = 30;
inline int Power(int a, int b)
{
    int ret = 1;
    for (; b; b >>= 1, a = 1LL * a * a % Mod)
        if (b & 1) ret = 1LL * ret * a % Mod;
    return (ret);
}

// ================================================================
const int MOD = 998244353;

struct mod_int {
    int val;

    mod_int(long long v = 0) {
        if (v < 0) v = v % MOD + MOD;
        if (v >= MOD) v %= MOD;
```

$Time\ Complexity = O(P * logP * logK)$ (because frequency polynomial is of size P not N)
$Space\ Complexity = O(P * logK)$

CHEF VIJJU'S CORNER 😃

More on Case of 0

The total number of sequences being N^k is obvious. For the other term - the sequences with no 0's, realize that we have $N - F[0]$ choices for each chosen element. Hence, for k chosen elements, our choices are

$(N-F[0])^k$

Why are we not using nCk formulas

This is very basic of counting. Generally, nCk formulas implicitly mean a given object can be chosen only once. So thats out.

But what about the nCk formulas where an object can be chosen multiple times? Even if they did work, thats a lot more work than using the basic fundamental "We have $N-F[0]$ choices for each index, hence $(N-F[0])^k$ choices for k elements." which is known to be correct.

Basic Funda of Generator/Primitive Root

Say we have a set containing numbers from 1 to $P-1$, and that P is prime. We call a number G generator of the set, if each of its power from 0 to $P-2$ modulo P correspond to a different number in the set. In other words, all the numbers in range 1 to $P-1$ can be written as powers of G.

And no, this is NOT going to be decimal or involve floating points. We can easily prove G is an integer, because G^1 is a power of G and it must represent one element of the set - hence G^1 is going to be one element in range 1 to $P-1$.

The main error people do while understanding this is that they forget this is modular arithmetic - even if G^k exceeds P, we are only interested in $G^k\%P$.

Why powers are done modulo P-1 instead of P

To know the answer - ask yourself, why are you thinking of doing it modulo P in first place? Because everything happens modulo P everywhere? Note that I am assuming P to be prime here.

You'd be surprised to know how basic and yet tricky it is. We, in fact, do powers modulo $P-1$ because everything happens modulo P.

By Fermat's little theorem, we know that $K^{P-1}\%P=1$. Now, say we want to find $K^X\%P$, where X is very large, in fact, larger than what you can store in 64-bit integer. For instance, say you know from problem that $X=6^N$ for N upto 10^9. So you have to find $K^{6^N}\%P$.

How do we do this? Fermat's little theorem to rescue! We know that $K^{P-1}=1$. Hence, we can rewrite X as $X=A*(P-1)+B$, and hence $K^X=K^{A*(P-1)}*K^B=(K^{P-1})^A*K^B=1^A*K^B=K^B$.

Hence we do modulo $P-1$ for powers.

- Say you have to calculate something like $K^{3^{3^N}}\%P$ where P is prime. How will you proceed?

On Polynomial used in All Possible Sums

The concept is actually simple and can be shown with example. Say you have an array $A=[3,3,3]$ and another array $B=[1,1]$. What are the possible sums and number of ways to get them? The possible sum is 4, and can be obtained in $3*2=6$ ways. Note that 2 ways are same if elements from same indices (for both arrays) are picked in both the ways. Hence its easy to see that number of ways is nothing but product of frequency.

Now, when we do a multiplication of these frequency polynomial - the powers of x add. This represents the sum which is possible. The coefficients of these powers (which are just the frequency of elements) multiply - which represent the number of ways to achieve them desired sum.

How to go about these questions involving FFT

Usually I see many people struggling to learn FFT. Why they struggle is because, they try to go directly into the math behind it. Well, its really intimidating.

Do you know how pros deal with FFT? Its simple - they dont. Most of people just make themselves aware on where to use it. Instead of equipping themselves with why it is correct (where many people get lost and give up), they equip themselves with where they can use it if given a problem.

As you see in setter's solution - many people generally use libraries and templates to solve these kind of questions, which is kind of good because you do not suffer from any bugs in implementation of those parts. That also stresses on the fact that they tackled the conept by knowing how to use it rather than why is it correct.

So if you are learning FFT, just know what it does, and where to use it. Skip those proof using complex numbers for now.

Setter's Notes

Expected Solution: We can rewrite each $A[i]$ as $A[i]\%P$. Now we have to handle the cases where the product becomes zero.

If there's at least one zero among the K numbers, the product becomes zero. So we can calculate this.

Now we delete all the zeros. In order to solve the problem we have to realize that multiplication is hard and we have to transform it into something like addition. We can find a primitive root G for P and rewrite $A[i] = X[i]$ such that $G^X[i] = A[i]$. Now multiplication in the previous problem becomes addition in this one. We add their powers when we multiply them.

So we are actually given a knapsack problem. We want to know in how many ways we can choose K numbers such that their sum modulo $(P-1)$ equals R for all Rs. Define a polynomial F such that $F[val]$ equals the number of vals in our new array. Now we can write F in point value form and raise each number to the Kth power and then rewrite F in polynomial form. This gives us the answer.

The only problem is that F should have a size of $N * P$. We can however fix this with a trick. Compute $T[1] = F$. Now compute $T[2] = T[1] * T[1]$. $T[2]$ should have twice the size of $T[1]$. When we computed $T[2]$, for each $i \geq P-1$ we can add the value of $T[2][i]$ to $T[2][i\%(P-1)]$ and only keep the first $P-1$ values. This way the size of $T[2]$ remains $P-1$. We can then compute $T[3], T[4], ..., T[log(K)]$. This requires $O(P * log(P) * log(K))$.

Now to get the answer we should multiply some of this polynomials (there are at most log(K) polynomials) with the same trick described above (reducing by half). This gives us a $O(P * log(P) * log(K))$ solution.

Related Problems

- Chef and Subsequences 6 - More on Multiplication to Addition trick
- Three Tower Coloring 2 -Fast Expo and Fermat's little theorem.
- POLYEVAL 3 - FFT
- BINOSUM 3 - FFT and Combinatorics
- CLGSUM 4 - FFT.

2

Reply

January Long Challenge 2020

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
CHOCOCHEF - Editorial editorial editorial start100	0
SQUAR - Editorial editorial editorial sieve start103	0
VIDEOWORTH - Editorial editorial editorial start109	0
GALACTICNW - Editorial editorial	0

There are 38 unread and 6 new topics remaining, or browse other topics in editorial

PROBLEM:

You are given a string S of '(', '*', and ')' and you choose a substring T uniformly at random. Process T from left to right. Whenever the number of closing parentheses in the current prefix is greater than the number of opening parentheses, change the closing parenthesis to an opening parenthesis. Find the expected number of changes.

SAMPLE EXPLANATION:

Some people were complaining about not having sample explanations, so here it is.
What does expected value mean?
Don't be lazy, did you try to google search it?
Don't be lazy, did you try to google search it?
Don't be lazy, did you try to google search it?

Expected value basically means an average value over all cases weighted according to the probability that the case occurs.

Since we choose T uniformly at random, meaning that each possible substring is equally likely, our average will be unweighted.

So, we need to find the average number of changes over all cases, and we can find the sum over all cases, and divide by the number of cases.

In the sample, $S =$ "((()))", and there are 15 possible substrings. I've listed the answer for each one of them:

```
"(" = 0
"(" = 0
"(" = 0
")" = 1
")" = 1
"((" = 0
"((" = 0
"()" = 0
"))" = 1
"(((" = 0
"(()" = 0
"())" = 1
"((()" = 0
"(())" = 0
"((()))" = 0
```

The sum is 4 and the average is $\frac{4}{15}$, so $\frac{4}{15}$ is our answer.

What does multiplicative inverse mean?"
Don't be lazy, did you try to google search it?
Don't be lazy, did you try to google search it?
Don't be lazy, did you try to google search it?

We can find the multiplicative inverse of Q with the formula $Q^{M-2} \mod M$ for any prime M (like $10^9 + 7$ in the problem).

For our final answer, first we calculate the multiplicative inverse of 15 with the formula above, which gives 466666670.

We multiply that by $P = 4$ modulo $10^9 + 7$ and we get the final answer of 866666673.

QUICK EXPLANATION 1

Use the contribution technique / linearity of expectation and find the number of subarrays which change i for each i. We can calculate $dp_{i,j}$ = the number of starting indexes which have $n_o - n_c = j$ after processing the first i elements. We can update this DP fast using lazy shifting techniques.

QUICK EXPLANATION 2

Let dp_i be the sum of answers for all subarrays starting at i. If $S_i =$ '(', find the first j which causes $n_o - n_c$ to be negative, and $dp_i = dp_j$. If $S_i =$ '*', $dp_i = dp_{i+1}$. Otherwise, find the first j which causes $n_o - n_c$ to be negative after we change S_i to '(', and $dp_i = dp_j + n + 1 - i$. Finding j for the transitions can be done in constant time by maintaining a stack of unmatched closing parentheses while processing from right to left.

EXPLANATION 1

Contribution Technique / Linearity of Expectation

Instead of finding the total sum of answers over all subarrays, let's focus on how much each index i contributes to the total sum. Index i's contribution to the total sum is the number of subarrays which cause i to change. To find the total sum, we can add up the contribution of all indexes instead.

Explained with Sample

In the sample, $S =$ "((()))", and only indexes 3 and 4 can contribute to the sum since those are the only closing parentheses.

We can check that the number of subarrays which cause index 3 to change is 2. Those subarrays are $[3, 3]$ and $[3, 4]$.

We can check that the number of subarrays which cause index 4 to change is 2. Those subarrays are $[2, 4]$ and $[4, 4]$.

We add up $2 + 2 = 4$, which is the total sum over all subarrays.

Check out the "related problems" section below.

Given that we want to use contribution technique on this problem, how do we find the number of subarrays which change a certain index i?

A subarray $[l, r]$ changes i if, when we process all characters from l to i (including changing closing parentheses to opening parentheses when necessary), $n_o - n_c$ is negative right before we perform any changes to i. Notice that as long as $r \geq i$, the choice of r does not affect whether i is changed or not.

Thus, we care about two things: the starting index l of a subarray and the value of $n_o - n_c$ when we reach index i.

This suggests a DP solution: $dp_{i,j}$ is the number of l such that the value $n_o - n_c$ is j after processing the first i characters.

We need to figure out the DP transitions, which involves some casework:

Transitions

If $S_i =$ '*':

$dp_{i,j} = dp_{i-1,j}$, since the value $n_o - n_c$ doesn't change.

We also create a new subarray $[i, i]$, so we should add 1 to either $dp_{i-1,0}$ before we transition or $dp_{i,0}$ after we transition.

If $S_i =$ '(':

$dp_{i,j+1} = dp_{i-1,j}$, since the value $n_o - n_c$ increases by 1.

We also create a new subarray $[i, i]$, so we should add 1 to either $dp_{i-1,0}$ before we transition or $dp_{i,1}$ after we transition.

If $S_i =$ ')':

$dp_{i,j-1} = dp_{i-1,j}$, since the value $n_o - n_c$ decreases by 1.

We also create a new subarray $[i, i]$, so we should add 1 to either $dp_{i-1,0}$ before we transition or $dp_{i,-1}$ after we transition.

We need to do something additional in this case: since $n_o - n_c$ might become negative (-1 to be precise), we may need to change the last closing parenthesis into an opening parenthesis. After we perform the change, $n_o - n_c$ will change from -1 to 1, so we need this additional step:

First, add $dp_{i,-1}$ to $dp_{i,1}$. Then, set $dp_{i,-1}$ to 0.

We have our DP transitions, but we still need to figure out how to calculate the answer. Remember that the criteria for i to change is that $n_o - n_c$ is negative (-1 to be precise) right before we perform any changes to i.

Thus, the number of ways to choose l for a subarray which changes i is $dp_{i,-1}$ before we change i. The number of ways to choose $r \geq i$ is $n - i$, so we should add $dp_{i,-1} \cdot (n - i)$ to the answer.

Our transition for the case $S_i =$ ')' is modified below:

Transition

After we have found the total sum, we find the expected value by dividing the sum by the number of subarrays, which is $\frac{n(n+1)}{2}$.

Sadly, this solution is $O(n^2)$, so let's try to find the bottleneck of this solution and improve it. Ideally, we want transitions to be constant time, but the shifting operations (such as $dp_{i,j} = dp_{i-1,j}$) take linear time.

Notice that the array basically remains the same after shifting, it's just the indexes which are changing. Also, we don't need the values of dp_{i-1} and all previous rows after we transition to i.

Let $dp_{i,j+o_i} = dp_{i-1,j}$. What if, we never actually calculate dp_i, but whenever we want to retrieve or modify the value $dp_{i,j}$, we actually just do it on $dp_{i-1,j-o_i}$?

We save the $O(n)$ transition and all other operations basically run in constant time.

Let's go further: We don't store dp_{i-1} as well, and if we want $dp_{i,j}$, we just take the value $dp_{i-2,j-o_i-o_{i-1}}$. Why not just delete all dp_i except for $dp_0 = d$, and whenever we want the value $dp_{i,j}$, we just look at d_{j+o} for some offset o? This allows to perform the shift transitions in constant time!

Using this trick, we shoule maintain an offset o while we iterate from left to right, and our new transitions look like:

Transitions

- Add 1 to d_o (which was $dp_{i-1,0}$).

If $S_i = $ '(':

- Subtract 1 from o to simulate $dp_{i,j+1} = dp_{i-1,j}$.

If $S_i = $ ')':

- Add 1 to o to simulate $dp_{i,j-1} = dp_{i-1,j}$.
- Add $d_{o-1} \cdot (n-i)$ (which was $dp_{i,-1} \cdot (n-i)$) to the answer.
- Add d_{o-1} to d_{o+1} (which was $dp_{i,1}$).
- Set d_{o-1} to 0.

In the implementation, in order to prevent negative indexes, d should be an array of size $2n+1$ and o should start from n.

EXPLANATION 2

Instead of doing a DP from left to right, let's try a DP from right to left. Let dp_i = the sum of answers for all subarrays starting at i. Our final answer is the sum of all dp_i divided by $\frac{n(n+1)}{2}$, the number of subarrays. We should do some casework to find the transitions:

Transitions

If $S_i = $ '*':

We can ignore S_i as it doesn't do anything, so $dp_i = dp_{i+1}$.

If $S_i = $ '(':

Let's somehow find the first index j such that the subarray $[i,j]$ contains more closing parentheses than opening parentheses.

By our definition of j, since it is the first such index, all indexes before j won't be changed and won't contribute to the answer.

Also, we know that in the subarray $[i,j-1]$, the number of the two types of parentheses must be equal. If there were more opening parentheses than closing ones, then the number of closing parentheses could not possibly have become greater after adding index j. If there were more closing parenthese than opening ones, this contradicts our assumption that j is the first index to have this property since $j-1$ also does.

Since all parentheses in $[i,j-1]$ cancel out, the indexes which are changed would be the same as if we started from index j instead. Thus, $dp_i = dp_j$.

If there is no such j, $dp_i = 0$.

If $S_i = $ ')':

S_i definitely will be changed, and all subarrays starting at i will cause i to change, so we add $n-i$ to dp_i.

Since we change S_i to '(', let's temporarily pretend that $S_i = $ '('. Then, we can perform the transition for $S_i = $ '(' as described above.

To complete the solution, we just need a fast way to find j in the transitions. Notice the similarity of this problem to parentheses matching: if all parentheses in $[i,j-1]$ cancel out, then they are balanced.

We should use a standard algorithm for parentheses matching [21] but process from right to left because that's the direction of our DP. So, we should maintain a stack of unmatched ')' s.

Whenever we encounter a '(', we should match it with the top of the stack (if the stack is empty, don't do anything). Then, as the top of the stack is the first unmatched ')', it is the first j which causes the number of closing parentheses to be greater than the number of opening ones.

PREREQUISITES:

Combinatorics, Fast Expontiation

PROBLEM:

Given an integer **N**, count the number of permutations that is increasing up to a certain point then deceasing after that.

EXPLANATION:

finding the formula

considering that this problem is requiring us to count something and that the constraints on N is very high this will hint us that the solution will be probably using combinatorics

let's say that we have an empty array of length **N** and we want to put the numbers from 1 to **N** in this array and satisfy the problem requirements, now let's say we want to add the number 1, we have **N** ways to put this number on the array but if we notice that we should not make 3 consecutive numbers such that the one in the middle is the minimum number among these 3 numbers then we will know that we cannot put the number 1 on any cell except the first and the last cells, because otherwise the number 1 will be in the middle of two bigger number eventually. So in conclusion we have only 2 possible ways to put the number 1, since we have put it in a border cell then we can imagine that we have an array of length **N-1** and now we want to put the number 2, again for the same reasons we have two ways and so on, until we want to put the number **N** we will only have 1 way instead of 2, so we have **N-1** number that have 2 ways to put, so by multiplication rule of combinatorics the answer is 2^{N-1}, finally we should subtract 2 from the answer because permutations 1 2 3 4 … **N** and **N N-1** … 3 2 1 should not be counted. we also should treat the case **N=1** alone because those two permutations become the same so we should subtract 1 instead of 2

to sum up, the answer is $2^{N-1}-2$ for all **N**>1 and 0 for **N**=1

algorithm for calculating the formula

an O(n) algorithm for calculating the formula would get the score of only the first sub-task so we need a faster way to compute it, fortunately there's O(log n) algorithm for it which is called "Exponentiation by squaring", you can find details about it here "Exponentiation by squaring - Wikipedia 4 "

the algorithm works by observing that $2^N = 2^{N/2} * 2^{N/2}$ if **N** is even, in this case we only need to recursively calculate $2^{N/2}$ once then squaring it. if **N** is odd it's very similar, we have $2^N = 2^{(N-1)/2} * 2^{(N-1)/2} * N$ so we only need to recursively calculate $2^{(N-1)/2}$ then square it then multiply it by **N**

since we always divide **N** by 2 then this algorithm takes only **O(log N)** steps

pseudo code:

```
power2(int N){
    if(N==0){
        return 1
    } else if(N is even){
        int H=power2(N/2)
        return H*H
    } else {
        int H=power2((N-1)/2)
        return H*H*2
```

```
        }
    }
```

SETTER'S SOLUTION

Can be found here [28] .

PREREQUISITES:

Combinatorics,Math,Number Theory

PROBLEM:

Consider an ordered 4 tree with **N** vertices. Your task is to calculate the expected value of the number of vertices having exactly **one** child in such tree assuming that it is uniformly chosen from the set of all ordered trees of size **N**.

Consider the answer to be a proper fraction **P/Q**, where **gcd(P, Q) = 1**. Then your task is to output two integers PQ^{-1} mod 10^9+7 and PQ^{-1} mod 10^9+9

EXPLANATION:

OBSERVATION 1:

The number of ordered trees consisting of **n** nodes is equal to the **(n-1)th catalan** 5 number. Imagine the **Depth-First-Search** order of the nodes of the trees, entering each node corresponds to an open bracket **(** , and finishing each node corresponds to a closed bracket **)** . So if we represented the **DFS** order of a tree it will form a simple balanced bracket sequence.

Two different ordered trees must have different bracket representations. So we can say that the number of ordered trees consisting of n nodes is equal to the number of balanced bracket sequences consisting of **(n-1)** pairs of brackets. **(n-1)** because our tree is **connected** so we must fix the root (the first and the last bracket in the expression). If the subexpression (the expression excluding the first and last bracket) is balanced, that guarantees that we will never exit the root before the last bracket and our tree is connected.

It's well known that the number of balanced bracket sequences of **n** pairs of brackets is equal to the **nth catalan** number.

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

Before reading the next observation you should be familiar with Linearity Of Expectation 9 .

OBSERVATION 2:

Let's consider all different **(n-1)** ordered trees, and think about applying this operation. Choosing an arbitrary node of a tree and linking it to a **single child (nth node)** and removing all edges between our chosen node and its children, then linking them to our new node. So our new node would serve as a single child of the chosen one.

A node having one child in a tree is equivalent to a pair of brackets, with a balanced expression between them and surrounded by a pair of brackets.

Observe that each node in a tree consisting of **n** nodes and having only **one** child can be formed by applying this operation to **only one** tree of **(n-1)** nodes. It's exactly the same as choosing an opened bracket and its corresponding closing bracket in a simple balanced bracket sequence framing the subexpression between by a pair of brackets (our new node).

Thus our nominator **P** would be

P = (number of trees of n-1 nodes * (n-1))

We multiplied by **n-1** which denotes the number of ways to choose the vertex we decided to apply this operation on.

Q = (number of trees of n nodes)

$$answer = \frac{C_{n-2} * (n-1)}{C_{n-1}}$$

According to the catalan number formula we wrote above, this can be reduced to:

$$answer = \frac{n * (n-1)}{4n-6}$$

This fraction can be reduced easily by calculating GCDs between the denominator and the nominator terms. We can calculate $\mathbf{PQ^{-1}}$ after calculating the modular inverse of **Q** considering each modulo.

Note:

It's true that this solution is given without a formal proof. But the idea is quite simple and correct. Actually its correctness can be proved by intuition. In real life contests you won't have the internet to read formal proofs and papers or search for formulas for a sequence. This solution is not hard to get from scratch and worths trying.

AUTHOR'S AND TESTER'S SOLUTIONS:

AUTHOR's solution: Will be found here 47

TESTER's solution: Will be found here 16

EDITORIALIST's solution: Will be found here 27

PRE-REQUISITES:

Combinatorics 1 , Finding ${}^{n}C_{k}\%p$ using Fermat's Little Theorem 3 , Data structures like multiset 2 .

PROBLEM:

Find number of ways to choose K out of N segments such that their common intersection is empty. Common intersection, in other words, means $L_1 \cap L_2 \cap ... \cap L_k = \phi$ $i.e.(empty/null)$. Print answer modulo $10^9 + 7$

QUICK EXPLANATION:

Key to AC- Its very easy if you have some practice in combinatorics, else the intuition may seem tricky. Finding number of violating cases was easier than finding number of good ones. Calculating answer as $Ans = Total\ Cases - Bad\ Cases$. Total cases, obviously, is ${}^{n}C_{k}$.

We pre-calculate the inverse and factorial of required numbers to calculate ${}^{n}C_{k}$ in $O(1)$ time. We can easily maintain a $multiset$ (to maintain order). We can easily formalize when all K elements intersect as if $min(R_1, R_2, .., R_{k-1}) \geq L_i$ where segment $\{L_i, R_i\}$ is the segment under consideration. After this, its simple calculation of ${}^{n}C_{k}$. We will discuss derivation under explanation section.

(WHAT? You want me to give away everything in Quick Explanation? xD.)

EXPLANATION:

This editorial will have a single section. Its assumed that you went through how to calculate ${}^{n}C_{k}$ as we wont be discussing this in detail in *official* part. We will simply see the intuition and derivation of formula. I will give whatever links I can for ${}^{n}C_{k}$ in Chef Vijju's Bonus Corner :). We will refer to @mgch (tester's) solution for implementation.

1. How to deduce that ans is calculated by finding number of bad combinations and subtracting them from total combinations?

This comes with practice. Really! It will seem something trivial to someone who is well versed with such questions. However, if you want what concrete steps we can analyze are-

- Easy formalization of condition when all K segments intersect.
- Total ways are easy to find. Simply ${}^{n}C_{k}$
- We will see below that a direct formula exists to find number of violating segments.

2. I have taken the input. What next?

Well, whats next? We solve the question! The very first thing to do is, we sort the segments. We sort the segments in **increasing order of L_i. If L_i are same, then the segment with larger R_i comes first.**

Why R_i in descending order? Simple, because if L_i are same, then inserting larger segment first helps us to easily find if k segments intersect. (Why?Q1)

Now we will maintain a multiset of lines. Multiset is used as it keeps them in sorted order. There are many implementations on what to store and how to use. Giving details of most easy one, I quote "we need not make a custom data type, merely storing the end points of lines can do." (Why?Q2) A hint is given in tab below.

Click to view

Focus on condition $min(R_1, R_2, .., R_{k-1}) \geq L_i$. What are we comparing? What things are hence, needed to be stored in set? Did we account of $L_1, L_2, ..L_{k-1}$ anywhere above? Is it necessary to hence, store $L_1, L_2...$?

The multiset will store the R_i in ascending order. Now, when do 2 horizontal lines intersect? Can you try framing conditions on when they interesect and when they dont?

Click to view

Obviously! When $R_1 \geq L_2$ where lines are $\{L_1, R_1\}$ and $\{L_2, R_2\}$. When will they not intersect hence? Easy to see now, if $L_2 > R_1$.

Now, we will follow a straightforward procedure-

1. For every segment $\{L_i, R_i\}$ do the following-
2. WHILE multiset is not empty, and the lines dont intersect- Delete the line with smallest R_i from multiset and check again.
3. Number of violating ways using this segment $\{L_i, R_i\}$ are ${}^{p}C_{k-1}$ where $p = size\ of\ multiset$
4. Insert end-point of this line in the set.

Lets discuss the intuition behind it. Step 1 is simple looping. Step 2, we discussed above when line intersect and when they dont. We need all k lines to intersect for a way to be violating. Hence, if $i^{'}th$ lines doesn't intersect, we delete the line with smallest R_i from multiset. Now, either the multiset is empty, or we have number of lines which intersect with given $i^{'}th$ line. (Why?Q3)

Now, what we have in multiset is a set of lines which intersect with $i^{'}th$ line. We **must** choose $i^{'}th$ line, and are free to choose rest $k-1$ lines from the multiset. If, thus, size of multiset is p, then number of ways of choosing $k-1$ lines is simply ${}^{p}C_{k-1}$. A simple code for above is given below-

```
multiset < int > f;
int bad = 0;
for (int i = 1; i <= n; ++i) {
    while (!f.empty() && *f.begin() < p[i].first) {
        f.erase(f.begin());
    }
    bad = (bad + 1LL * comb(f.size(), k - 1)) % MOD;//comb(a,b)=aCb
    f.insert(p[i].second);
}
```

SOLUTION:

The codes which I received are pasted below for reference. This is done so that you dont have to wait for @admin to link solutions up :-). Please copy them and paste at a place where you are comfortable to read :).

Pre-requisites:

Combinatorics and some maths

Statement Abstraction

Let s to be *"abaabbaaaabbbb..."* and $f(x)$ to be number of possible palindrome substring segments in $s[1 \dots x]$. Calculate $\sum_{i=1}^{n} f(i) \bmod 10^9 + 7$.

Explanation

Let $f(x) = o(x) + t(x)$, where $o(x)$ is the number of single-charactered palindromes(*"aaaa"*, *"bb"*, etc) and $t(x)$ is the number of three-charactered palindromes(*"aaabbbbaaa"*, etc).

The first key point is, there is no other kind(except for two types described above) of palindrome substrings possible, because there is no two same segments in whole s. The second key point is, we can separate $o(x)$ and $t(x)$ in terms of calculation because they are independent.

Let's assume following situation: s = *"...aa b...b aa...aa bb...bb"*, where each segment's end index are ppp, pp, p, and x. ($s[\dots ppp]$ is *"...aa"*, $s[ppp+1 \dots pp]$ is *"b...b"*, $s[pp+1 \dots p]$ is *"aa...aa"*, and $s[p+1 \dots x]$ is *"bb...bb"*) Then we can construct following formula:

- $o(i) = o(p) + \binom{i-p+1}{2}$, because new cases on last segment selection is added.
- $t(i) = t(p) + \min(pp - ppp, i - p)$, because new cases on *"bb...aa...bb"* is added.

But our goal is to calculate sum of $f(x)$. So let's make following formula:

- $\sum_{i=p+1}^{x} o(i) = o(p) \cdot (x-p) + \sum_{i=1}^{x-p} \binom{i+1}{2} = o(p) \cdot (x-p) + \frac{1}{6} \cdot (x-p) \cdot (x-p+1) \cdot (x-p+2)$.
- $\sum_{i=p+1}^{x} t(i) = t(p) \cdot (x-p) + (1 + 2 + \dots + R_1) + R_2 = t(p) \cdot (x-p) + \frac{1}{2} \cdot R_1 \cdot (R_1+1) + R_2$. Value of R_1 and R_2 depends on which ($pp - ppp$ and $x - p$) is bigger.

So in each segment, you calculate following formulas to get sum of f value in segments and last f value of segment, then recursively pass to next segment until you reach index n. You have maximum $\log n$ segments, so time complexity is $O(\log n)$ per test case.

PREREQUISITES:

combinatorics, fast-fourier transform, divide and conquer

PROBLEM:

Given an array of integers $A_1, A_2, ..., A_N$ and an integer K, count number of ways to choose a multiset(*i.e.* a set with duplicate elements) of size K. Two multisets are different iff count of any element is different in both.

QUICK EXPLANATION:

==========

Answer is coefficient of x^K in polynomial $\frac{(1-x^{f_1+1}) * (1-x^{f_2+1}) * .... * (1-x^{f_k+1})}{(1-x)^k}$, where $f_1, f_2, ..., f_k$ are frequencies of k distinct elements present in the array A. Numerator of this polynomial can be evaluated using any fast polynomial multiplication algorithm since it's degree is N.

EXPLANATION:

==========

Theorem 1:
It's a popular result using generating functions that number of integral solutions to equation $x_1 + x_2, ..., +x_r = N$, where $x_i \leq l_i$ are coefficient of x^N in $(1 + x + x^2 + ... + x^{l_1}) * 1 + x + x^2 + ... + x^{l_2}) * ... * 1 + x + x^2 + ... + x^{l_r})$, which we re-write as $\frac{(1-x^{l_1+1}) * (1-x^{l_2+1}) * .... * (1-x^{l_r+1})}{(1-x)^r}$.

Theorem 2:
Coefficient of x^k in $(1-x)^{-N}$ is $C(N+k-1, k)$, where $C(i,j)$ is number of ways to choose j distinct elements from a set of i distinct elements. This can be proved using Taylor series expansion.

We'll avoid the proof here, but you can follow above link to explore more.

Now, assume we have k distinct elements in the array with frequencies $f_1, f_2, ..., f_k$. Let's say x_i is the frequency of i^{th} distinct element in chosen multiset, then $x_1 + x_2 + ... + x_k = K$, where $x_i \leq f_i$. So, from above theorem, we need to find coefficient of x^K in polynomial $\frac{(1-x^{f_1+1}) * (1-x^{f_2+1}) * .... * (1-x^{f_k+1})}{(1-x)^k}$. Note we can evaulate the numerator since it's degree is $O(f_1 + f_2 + ..., f_k) = O(N)$. Let's say we use FFT to multiply two polynomials of degree p in $O(p \log p)$. If we start multiplying these k terms in numerator in the given order, it could still take $O(N^2)$ worst case. Here comes in the idea of divide and conquer. At each step we break down numerator into two parts with $k/2$ terms each, solve them recursively and multiply to find the polymial. This approach takes $O(N \log^2 N)$ worst case.

For multiplication of two polymials you can use FTT or even Karatsuba. For further reading on these algorithms you can read this 11 .

PREREQUISITES:

Heavy-light Decomposition, Inclusion-Exclusion, Combinatorics and Segment Tree.

PROBLEM:

Given a tree with N nodes where each node is assigned colors Black and White, denoted by 1 or 0 respectively, we have to answer Q queries where each query specifies a path, say from node L to R and we have to count the number of ways to choose a set of three distinct nodes such that

- The chosen nodes lie on the shortest path between node L and R
- For any two chosen nodes, say u and v, there is at least one black colored node on the shortest path between nodes u and v (both inclusive).

DEFINITIONS

- For a query (L, R), a triplet (u, v, w) represent the set of nodes u, v and w such that all these nodes lie on shortest path from L to R and while moving from L to R, first u is found, then v and then w.
- Let $P(u, v)$ denote the path from u to v
- A triplet (u, v, w) is valid for a query (L, R) if there's at least one black node on $P(u, v)$ and at least one black node on $P(v, w)$

QUICK EXPLANATION

- The total number of triplets is $\binom{K}{3}$ where K is the number of nodes on the path from L to R. The mutually disjoint exhaustive set of invalid triplets are as follows.
  - The triplet has no black node on $P(u, w)$. The number of such triplets is given by $\sum_{x \in S} \binom{x}{3}$ where S contain blocks of consecutive 0s on $P(L, R)$
  - The triplet has no black node on $P(u, v)$ and at least one black node on $P(v, w)$
    OR
    The triplet has at least one black node on $P(u, v)$ and no black node on $P(v, w)$
    The number of such triplets is given by $\sum_{x \in S} (K - x) * \binom{x}{2}$
    We shall exclude these invalid triplets from total triplets to get the number of valid triplets.
- We need to maintain this information over ranges, so we need to use the segment tree, each node storing the above information, and handle node merging while maintaining the same set of information.
- In order to extend the above solution over a tree, we need to apply the Heavy-light Decomposition.

EXPLANATION

Let's solve a simpler problem.
Given an array A of length K consisting of 0 and 1 only, answer queries to count the number of valid triplets for specified subarray.

Now, let's count the number of invalid triplets. For a triplet (u, v, w) to be invalid, we need either $A[u, v]$ to contain all zeroes, or $A[v, w]$ to contain all zeroes or both.

If we group the consecutive 0s and 1s, we can see that an invalid triplet contains at least two of the three chosen nodes in the same group. Let's see an example.

Consider $K = 15$ and array `1 1 1 0 0 0 0 0 1 1 1 1 0 0 0`
Grouping 0s in the above array, we get One group of 5 zeroes and one group of 3 zeroes. So multiset $S = \{5, 3\}$

Let's count invalid triplets (u, v, w) where there's no 1s in $A[u, w]$. We can easily see that we need u, v and w to be in the same group which is the same as the number of ways to select three items from x elements, which is $\binom{x}{3}$ where x is group size. Hence, the number of such triplets is $\binom{5}{3} + \binom{3}{3}$ which is $10 + 1 = 11$

Now, let us count the number of invalid triplets where

- $A[u, v]$ contain all 0s and $A[v, w]$ contain at least one 1
- $A[u, v]$ contain all 1s and $A[v, w]$ contain all 0s

Now, we need either (u, v) to lie in the same group of 0s and w to be not in the same group (which implies at least one 1, OR (v, w) to lie in one group and u to be not in the same group.

The number of ways to choose a pair of nodes in the same group is $\binom{x}{2}$ and we have $K - x$ choices for third node where x is group size. Hence, the total number of invalid triplets of this type is $(K - x) * \binom{x}{2}$. In current example, it becomes $(15 - 5) * \binom{5}{2} + (15 - 3) * \binom{3}{2} = 100 + 36 = 136$

Hence, the total number of invalid triplets is $11 + 136 = 147$. Total number of triplets is $\binom{15}{3} = 455$

Hence, the number of valid triplets is given by 308.

In general, the number of valid triplets becomes $\binom{K}{3} - \sum_{x \in S} [(K - x) * \binom{x}{2} + \binom{x}{3}]$

Now, we know how to calculate the valid triplets, and It is intuitive to use range Data structure like segment tree, but what information to store to answer subarray queries?

A node in segment tree corresponds to a range, and it is also possible for a group to lie in different nodes, so in each node, we need to store whether whole range contains 0s only and if not, the number of 0s at start of range, the number of 0s at end of range.

Now, Ignoring above, we also need to be able to calculate $\binom{x}{2}$, $x * \binom{x}{2}$ and $\binom{x}{3}$ for all groups of 0s lying completely inside this range. Turns out this information is enough.

We just need to merge the two ranges carefully taking care of prefix and suffix 0s and when they are merged.

This allows us to answer subarray queries in $O((Q + K) * log(K))$ time.

Now, returning to our original problem, we have done all the hard work, we just need to extend this over paths in the tree and as you guys familiar with this type of problem may have guessed, we use Heavy Light Decomposition 6 .

Stating briefly from the above link, The essence of heavy light decomposition is to **split the tree into several paths** so that we can reach the root vertex from any v by traversing at most logN paths. In addition, none of these

paths should intersect with another. As we can see here, we have exactly done that here, solved the same problem for array and using HLD to extend it over the tree.

Implementation things to take care, if you don't want to go crazy debugging:

- While merging nodes, be careful not to include the LCA node twice.
- Directions matter here. If a node represents information for $P(u, v)$ then we need to flip start and end block information to get information for $P(v, u)$. This is needed while merging paths.
- Same way, the merging is non-commutative. That matters in the segment tree.

While the above solution is the intended solution, there also exists a solution without using Heavy Light Decomposition, relying on precomputation, as used by the tester, as well as by most of the users.

TIME COMPLEXITY

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

Prerequisite :- Recursion, Permutation and Combinations

Reduced Problem Statement :-
For given N distinct numbers i.e 1…N there are in total N! possible permutations, out of these permutations you have to find Kth smallest permutation.
Note - Permutation means each of several possible ways in which a set or number of things can be ordered or arranged.

Explanation :-

There are 2 major ways to solve this type of problem. They are :

1. Apply brute force and find all possible permutations and then sort these permutations according to their numeric value and print the Kth smallest permutation.
2. Apply some kind of Recursive algorithm that will help to find the number at a particular position such that all constraints are satisfied.

The first approach is not feasible for Pen and Paper-based exams because it will take a huge amount of time, for example, for N=6 there are 720 possible permutations, you will have to write all 720 permutations and then sort these permutations for finding Kth smallest permutation. (However, this approach is feasible to solve using computer as constraints for this problem is 2<= N <= 9). So let us discuss the second approach in detail with this question in mind. Later you can apply this approach to solve other questions too.

Recursive Approach :-

This approach is kind of Mathematical type, so some basic facts that will be used to solve this problem are :-

1. The total number of permutations formed by N distinct integers is N!
2. The total number of permutations formed by N distinct integers after fixing the first character is (N-1)!. Similarly, if we fix the first i characters, and there are (n-i) vacant positions, then the number of possible permutations is (n-i)!.

In this approach, we will find the character that will be present in our Kth smallest permutation at position i.
First, we have to understand that if we fix character at first i positions then there are (n-i)! Possible permutations.
Hence for each character at the ith position, there are (n-i)! Permutations, for example, if N was 5 and we currently have 2 3_ _ _ (where _ represents unknown characters) then the remaining characters are 1,4,5 so we have 3! = 6 in total.
So using K we will find what character Kth permutation will have at the ith position.
Hence after fixing the ith character, there are (n-i) positions and hence (n-i)!

Hence if permutations are sorted then -

- The first smallest character will be in first (n-i)! Permutations,
- The second smallest character will be in second (n-i)! Permutations, and so on…

In short, the jth character will be from (j-1)*((n-i)!) + 1 to (j*(n-i)!) permutation, also we want Kth permutation (we will maintain K i.e K<= (n-i+1)!, I will explain it in the next paragraph), so character at ith position will be ceil (K/(n-i)!).
(Note - the ceil function returns the smallest integer that is greater than or equal to x ie: rounds up the nearest integer.)
So now we have got our character for ith position now we know that for ((j-1)*(n-i)! + 1) to (j*(n-i)!) smallest permutations j will be present at ith position, hence ((j-1)*(n-i)! + 1) <= K <= (j*(n-i)) and we cannot use j for further positions.
We will now find the character at (i+1) position, but for that, we need to update K because we need to narrow down our range hence our new K will be K' = K - ((j-1)*(n-i)!), and this step of finding character at ith position will be done recursively.
Let find(N, K, i, set of available character) be a function that will find the character at the ith position, hence its definition will be:

find(N,K,i,set of available characters){

```
    if( i>N )
          return;
    J = ceil ( K/( N-i )! )
    Print Jth smallest character from set of available character
    Delete Jth smallest character from set
    K = K - ( J-1 )*( N-i )!
    find( N,K,i+1,set of available character )

}
```

Solving Sample test case :-

N = 3 , K = 4 , S = { 1,2,3 }
Call for find(3,4,1,{ 1,2,3 })

1. I = 1
   J = ceil(K/(N-i)!) = ceil(4/(3-1)!) = ceil(4/(2)!) = ceil(4/2) = 2
   Jth smallest character available from S is 2
   Hence character at 1st position is 2
   Update S to { 1,3 }
   K = K - (J-1)*(N-i)! = 4 - (2-1)*(3-1)! = 4 - 2 = 2*
   Call for find(3,2,2,{ 1,3 })

2. I = 2
   J = ceil(2/(3-2)!) = ceil(2/1) = 2
   Jth smallest character available from S is 3
   Hence character at 2nd position will be 3
   Update S to { 1 }
   K = K - (J-1)*(N-i)! = 2 - (1)*(3-2)! = 2-1 = 1*
   Call for find(3,1,3,{ 1 })

3. I = 3
   J = ceil(1/(3-3)!) = ceil(1/1) = 1
   Jth smallest character available from S is 1
   Hence character at 3rd position will be 1
   K = K - (J-1)*(N-i)! = 1 - (0)*(0)! = 1*
   Call for(3,1,4,{})

Hence answer for N=3 and K = 4 will be 2 3 1

Solving Subtasks :-

1) N = 5, K = 76
Solution:
Answer = _ _ _ _ _

1. N = 5, K = 76, I = 1, S = { 1,2,3,4,5 }
   J = 4
   Answer = 4 _ _ _ _
   K = 76 - 72 = 4
   S = { 1,2,3,5 }
   I = I + 1 = 2
2. N = 5, K = 4, I = 2 , S = { 1,2,3,5 }
   J = 1
   Answer = 4 1 _ _ _
   K = 4 - 0 = 4
   S = { 2,3,5 }
   I = I + 1 = 3
3. N = 5, K = 4, I = 3, S = { 2,3,5 }
   J = 2
   Answer = 4 1 3 _ _

K = 4 - 2 = 2
S = { 2,5 }
I = I + 1 = 4

4. N = 5, K = 2, I = 4, S = { 2,5 }
   J = 2
   Answer = 4 1 3 5 _
   K = 2 - 1 = 1
   S = { 2 }
   I = I + 1 = 5
5. N = 5, K = 1, I = 5, S = { 2 }
   J = 1
   Answer = 4 1 3 5 2
   K = 1 - 0 = 1
   S = {}
   I = I + 1 = 6

Hence answer is 4 1 3 5 2

2) N = 7 , K = 4197
Solution :
Answer: _ _ _ _ _ _ _

1. N = 7, K = 4197, I = 1, S = { 1,2,3,4,5,6,7 }
   J = 6
   Answer = 6 _ _ _ _ _ _
   K = K - 5*720 = 597
   S = { 1,2,3,4,5,7 }
   I = I + 1 = 2

2. N = 7, K = 597, I =2, S = { 1,2,3,4,5,7 }
   J = 5
   Answer = 6 5 _ _ _ _ _
   K = K - 4*120 = 117
   S = { 1,2,3,4,7 }
   I = I + 1 = 3

3. N = 7, K = 117, I = 3, S = { 1,2,3,4,7 }
   J = 5
   Answer = 6 5 7 _ _ _ _
   K = K - 4*24 = 21
   S = { 1,2,3,4 }
   I = I + 1 = 4

4. N = 7, K = 21, I = 4, S = { 1,2,3,4 }
   J = 4
   Answer = 6 5 7 4 _ _ _
   K = K-3*6 = 3
   S = { 1,2,3 }
   I = I + 1 = 5

5. N = 7, K = 3, I = 5 , S = { 1,2,3 }
   J = 2
   Answer = 6 5 7 4 2 _ _
   K = K - 1*2 = 1
   S = { 1,3 }
   I = I + 1 = 6

6. N = 7,K = 1,I = 6,S = { 1,3 }
   J = 1
   Answer = 6 5 7 4 2 1 _
   K = K - 0*1 = 1

S = { 3 }
I = I + 1 = 7

7. N = 7, K = 1,I = 7, S = { 3 }
   J = 1
   Answer = 6 5 7 4 2 1 3
   K = K - 0*1 = 1
   S = { }
   I = I + 1 = 8

Hence Answer is 6 5 7 4 2 1 3

3) N = 9 , K = 191082
Answer = _ _ _ _ _ _ _ _ _

1. N = 9,K = 191082, I = 1, S = { 1,2,3,4,5,6,7,8,9 }
   J = 5
   Answer = 5 _ _ _ _ _ _ _ _
   K = K - 4*40320 = 29802
   S = { 1,2,3,4,6,7,8,9 }
   I = I + 1 = 2

2. N = 9,K = 29802, I = 2, S = { 1,2,3,4,6,7,8,9 }
   J = 6
   Answer = 5 7 _ _ _ _ _ _ _
   K = K - 5*5040 = 4602
   S = { 1,2,3,4,6,8,9 }
   I = I + 1 = 3

3. N = 9,K = 4602, I = 3, S = { 1,2,3,4,6,8,9 }
   J = 7
   Answer = 5 7 9 _ _ _ _ _ _
   K = K - 6*720 = 282
   S = { 1,2,3,4,6,8 }
   I = I + 1 = 4

4. N = 9, K = 282, I = 4, S = { 1,2,3,4,6,8 }
   J = 3
   Answer = 5 7 9 3 _ _ _ _ _
   K = K - 2*120 = 42
   S = { 1,2,4,6,8 }
   I = I + 1 = 5

5. N = 9, K = 42 , I = 5 , S = { 1,2,4,6,8 }
   J = 2
   Answer = 5 7 9 3 2 _ _ _ _
   K = K - 1*24 = 18
   S = { 1,4,6,8 }
   I = I + 1 = 6

6. N = 9, K = 18, I = 6, S = { 1,4,6,8 }
   J = 3
   Answer = 5 7 9 3 2 6 _ _ _
   K = K - 2*6 = 6
   S = { 1,4,8 }
   I = I + 1 = 7

7. N = 9, K = 6, I =7, S = { 1,4,8 }
   J = 3
   Answer = 5 7 9 3 2 6 8 _ _
   K = K - 2*2 = 2

S = { 1,4 }
I = I + 1 = 8

8. N = 9,K = 2, I = 8, S = { 1,4 }
   J = 2
   Answer = 5 7 9 3 2 6 8 4 _
   K = K - 1*1 = 1
   S = { 1 }
   I = I + 1 = 9

9. N = 9, K = 1, I = 9,S = { 1 }
   J = 1
   Answer = 5 7 9 3 2 6 8 4 1
   K = K - 0*1 = 1
   S = { }
   I = I + 1 = 10

Hence Answer is 5 7 9 3 2 6 8 4 1

Bonus Question :-

In the above problem, you were given N distinct integers but what if you are given N integers but they are not distinct for example N = 7, S = { 1,1,1,1,2,2,3 } and K = 1029

PREREQUISITES :

Number Theory, Inclusion Exclusion Formula, Basic Combinatorics

PROBLEM :

Given 2 integer N and K, we need to find the summation of the GCD of each sub sequence of length K of the sequence $1, 2, ...N$. We need to keep K fixed, and find the answer for each N in the range $1, 2, ...200000$

QUICK EXPLANATION :

The expected value of the number to be calculated shall be $\frac{X}{\binom{N}{K}}$. We now consider the subtask of trying to find X.

The number of numbers in the range $1, 2, ...N$ that are divisible by some number i is $\lfloor \frac{N}{i} \rfloor$, so the number of sequences of length K whose GCD is divisible by i is $\binom{\lfloor \frac{N}{i} \rfloor}{K}$. Then, we can use the inclusion exclusion method to find the number of sequences whose GCD is divisible by i and at least one prime even after all numbers are divided by i. We can speed up this inclusion exclusion using the Mobius Function. This will help us get the number of sequences whose GCD is exactly i. This gives us an efficient method to calculate the answer for a number N in $O(\sum_{i=1}^{N} divisors(i))$ time. Then, we can prove that we can calculate the answer for some N from the answer $N-1$ in $O(divisors(N))$ time by using the fact that $\lfloor \frac{N}{i} \rfloor \neq \lfloor \frac{N-1}{i} \rfloor$ if and only if i is a divisor of N.

So, we can overall calculate the answer over all N in $O(\sum_{i=1}^{N} divisors(i))$ time.

EXPLANATION :

Keeping N fixed, let's see how we can solve the problem :

The expected value of the number to be calculated shall be $\frac{X}{\binom{N}{K}}$. We now consider the sub task of trying to find X.

Let's try and invert the given subtask a little bit. Instead of finding the summation of GCD's of each subsequence of length K, we' try and find $\forall i \;\; 1 \leq i \leq N$, the number of subsequences of length K having GCD equal to i. For some i, let's call the so found number $f(i)$.

Then, we can calculate X as $\sum_{i=1}^{N} i \cdot f(i)$. The main part of this problem now boils down to calculating $f(i)$.

For some i, the number of numbers in the range $1, 2, ...N$ divisible by i is exactly $\lfloor \frac{N}{i} \rfloor$. These numbers are $1 \cdot i$, $2 \cdot i$, ... $\lfloor \frac{N}{i} \rfloor \cdot i$

So, assume we want to find the number of subsequenes whose GCD is divisible by i. That number shall be $\binom{\lfloor \frac{N}{i} \rfloor}{K}$. If we select any subsequence of length K consisting of only multiples of the number i, then it's obvious , the GCD of any such subsequence is also divisible by i. Let's call this $G(i)$

Now, we an deduce that if the GCD of a subsequence is divisible by i, then the GCD of the subsequence is a multiple of i. . Now, If we subtract from $G(i)$, the number of subsequences of length K such that their GCD is a multiple of i but $\neq i$. Then we can easily get $f(i)$ from $G(i)$.

So, we need to calculate the number of sub sequences of length K, such that their GCD is divisible by i and is also divisible by at least one prime number even after dividing by i.

Now, you need to read about the Mobius Function 78 here and come back. We make a small change to that function, let $\mu(1) = 0$. The Mobius function is a faster way of performing inclusion exclusion.

Mathematically speaking , we get :

$$f(i) = G(i) + \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} \mu(j) \cdot G(i \cdot j)$$

And the number X is :

$X = \sum_{i=1}^{N} i \cdot f(i)$, So

$$X = \sum_{i=1}^{N} i \cdot G(i) + \sum_{i=1}^{N} \sum_{j=1}^{\lfloor \frac{N}{i} \rfloor} i \cdot \mu(j) \cdot G(i \cdot j)$$

We can find the first part of the summation easily. So, the second part can be inverted to :

$\sum_{z=1}^{N} G(z) \cdot \sum_{d|z} d \cdot \mu(\frac{z}{d})$.

And finally, we get

$$X = \sum_{i=1}^{N} i \cdot G(i) + \sum_{i=1}^{N} G(i) \cdot \sum_{d|i} d \cdot \mu(\frac{i}{d})$$

If we precompute the divisor of each function and the mobius function, then we know that for a fixed z , the sum $\sum_{d|z} d \cdot \mu(\frac{z}{d})$ can be calculated easily. We know $G(i) = (\lfloor \frac{N}{i} \rfloor_{i})$.

So, after some pre computations, we can use the above formula to calculate the answer for fixed N easily in $O(N)$

The last part we still have to deal with is calculating the answer for each N in the range $1, 2, ...200000$. If we can precompute each of these too, then all queries can be answered in $O(1)$.

Let's call for some z, $\sum_{d|z} d \cdot \mu(\frac{z}{d}) = sum[z]$. This number remains unaltered regardless of N. Now, we prove by induction that if we have calculated the answer for $N-1$, then we can calculate the answer for N in $O(divisors(N))$ time.

We can re-write X as :

$X = \sum_{i=1}^{N} i \cdot G(i) + \sum_{i=1}^{N} G(i) \cdot sum[i]$. So,

$$X = \sum_{i=1}^{N} (i + sum[i]) \cdot G(i).$$

A crucial observation we make here is that the value $\lfloor \frac{N}{i} \rfloor \neq \lfloor \frac{N-1}{i} \rfloor$ if and only if N is a multiple of i.

So, for some i, $G(i)$ changes only at values of N when N is a multiple of i. So, if we know the answer for $N-1$, then we can use its answer, and replace $G(i)$ for each divisor of N to find the new answer.

There are many many problems before that can be used to learn how to compute the Mobius Function, the divisors of a number and all factorials and inverse factorials fast. You need to consider these to be pre-requisites, as they are quite common by now.

That's it, Thank you

Your comments are welcome !

COMPLEXITY ANALYSIS :

Time Complexity : $O((\sum_{i=0}^{MaxN} divisors[i]) + Q)$, where $MaxN \approx 200000$

Space Complexity : : $O(\sum_{i=0}^{MaxN} divisors[i])$, where $MaxN \approx 200000$

PREREQUISITES:

Basics of polynomial expansion, binomial coefficients [5] , modular exponentiation [6]

PROBLEM:

Let P be a polynomial in n variables given by:

$$P(x_1, x_2, \ldots, x_n) = (k + x_1 + x_2 + \cdots + x_n)^m$$

Here n, k and m are positive integers. Find the sum of all coefficients of the terms of the polynomial which have even powers in each of the n variables and output it modulo $10^9 + 7$.

QUICK EXPLANATION:

Find the sum of all values of $P(x_1, x_2, \ldots, x_n)$ where $x_i \in \{-1, 1\}$ and divide it by 2^n to find the answer. Find this in $O(n \log m)$ by considering how many times each value repeats.

EXPLANATION:

Let's first solve the problem for $n = 1$. We can divide the terms of the polynomial in 2 categories, terms with even powers and terms with odd powers. We observe that $P(1)$ is the sum of all coefficients of terms of the polynomial and $P(-1)$ is the sum of coefficients of terms with even powers subtracted by the sum of coefficients of terms with odd powers. Thus the required sum will be:

$$\frac{P(1) + P(-1)}{2}$$

Now let's try to generalize this for any n. Say we divide the terms of the polynomial in 2^n categories according to the parities of the powers of the n variables

For a particular value of $P(x_1, x_2,, x_n)$ where $x_i \in \{1, -1\}$, we see that terms of a particular category are either added or subtracted depending on whether the number of variables having odd power in the term and having value $= -1$ are even or odd respectively.

Consider the sum S of all 2^n values of $P(x_1, x_2, \ldots, x_n)$ where $x_i \in \{1, -1\}$. Observe that the terms with even powers in all the variables will always be added. Thus, they will appear 2^n times. For terms of any other category, say a category with $i > 0$ variables having odd powers, the terms will be added if j of these i terms have value -1 where j is a even number. Thus, the terms will be added A times where,

$$A = \sum_{2j \le i} \binom{i}{2j}$$

And the terms will be subtracted B times where,

$$B = \sum_{2j+1 \le i} \binom{i}{2j+1}$$

But we know

$$\sum_{2j+1 \le i} \binom{i}{2j+1} = \sum_{2j \le i} \binom{i}{2j} = 2^{i-1}$$

This has various proofs 14 .

Thus, these terms will be added and subtracted equal number of times and the final sum S will just be equal to 2^n times the sum of coefficients of all terms with even powers in all the variables. Our required sum will then be $\frac{S}{2^n}$.

We can find S in $O(n \log m)$ by observing that the value $(k + n - 2i)^m$ appears $\binom{n}{i}$ times.

SOLUTIONS:

Commented Code 40

This was my first time setting a problem, so I hope you enjoyed it. I would love to hear your thoughts on it.

PREREQUISITES:

Math, Combinatorics

PROBLEM:

Find the number of ways to divide a set of size N into even and odd sized non empty subsets.

QUICK EXPLANATION:

The answer is there are pow(2,N-1)-1 and pow(2,N-1) even and odd sized non empty subsets

EXPLANATION:

Considering (N,r) for r < 0 and r > N is 0,
the number of even sized subsets is sum of (N,2*r) for r running from 0 to N/2, and the number of odd sized subsets is sum of (N,2*r+1) for r running from 0 to N/2.
The sum of (N,2*r) for r running from 0 to N/2 is pow(2,N-1).
The sum of (N,2*r+1) for r running from 0 to N/2 is also pow(2,N-1).
These are the popular results from combinatorics and can be verified from here 2 and here 1 .
As we don't need empty subset,
the answer for even sized subsets is pow(2,N-1)-1.
the answer for odd sized subsets is pow(2,N-1)-1.

The answer for N = 0 is 0 for both even and odd sized subsets .

SOLUTIONS:

If unclear on this, refer example

Consider first set $[a, b, c]$ and second set $[d, e, f]$ where a, b, c, d, e, f denote the indices.
We have $3! = 6$ ways to make pairs

- $(a, d), (b, e), (c, f)$
- $(a, d), (b, f), (c, e)$
- $(a, e), (b, d), (c, f)$
- $(a, e), (b, f), (c, d)$
- $(a, f), (b, d), (c, f)$
- $(a, f), (b, e), (c, d)$

Lastly, for each of the above way, we have considered pair (p, q) to be same as (q, p), which is not the case in our problem, so each pair can appear in two ways giving 2^j ways for each above pairings.

If the next group is palindrome
Assume a palindrome string appearing x times, we can select j strings to appear in left half in ${}^{x}C_{j}$ ways, j strings from remaining $x - j$ strings to appear in right half in ${}^{x-j}C_{j}$ ways, and the number of ways to make pairs is $j!$
(Factor 2^j does not appear here, since both sets are chosen from the same set of x strings, and not different sets)

So, we have $D_{(i,sz,0)} = \sum_{j=0}^{2*j \le x, j \le sz} DP_{(i-1,sz-j,0)} * {}^{x}C_{j} * {}^{x-j}C_{j} * j!$

Now, for $DP_{(i,sz,1)}$, Either previous we have chosen middle string one of the previous $i - 1$ groups, or we choose middle in this group.

Transition with not choosing middle string from current group is same in idea as for $DP_{i,sz,0}$ If middle is chosen from this group, it's index can be chosen in x ways, and after that, j indices to appear in left half can be chosen in ${}^{x-1}C_{j}$ ways and then we can choose indices to appear in right half in ${}^{x-j-1}C_{j}$ ways. Making pair for each way of choosing is $j!$, giving us the following recurrence for $DP_{(i,sz,1)}$

$$DP_{(i,sz,1)} = \sum_{j=0}^{2*j \le x, j \le sz} DP_{(i-1,sz,1)} * {}^{x}C_{j} * {}^{x-j}C_{j} * j! + x * DP_{(i-1,sz-j,0)} * {}^{x-1}C_{j} * {}^{x-1-j}C_{j} * j!$$

That's it, computing the number of ways this way, if n is the number of groups, the final answer is $\sum_{j=0} DP_{(n,i,0)} * i! + DP_{(n,i,1)} * i!$ (Since left indices can also appear in any order, and exclude the empty subsequence.

This solution is of the complexity of order $O(n * M^2)$ since for each state, time taken is $O(M)$ and there are $(n * M)$ states though this solution runs fast in practice since the sum of degrees of P and$Q$ does not exceed $2 * M$.

Speedup using FFT/NTT

Let's rearrange states of DP to be (i, odd, sz) and assume $DP_{(i,odd)}$ represent a polynomial where coefficient of x^{sz} in this polynomial denote $DP_{(i,sz,odd)}$

For a non-palindrome group withfrequency of string and it's reverse being x and y, we can write another polynomial $P(x) = p_0 + p_1 * x \dots p_{min(x,y)} * x^{min(x,y)}$ where $p_j = {}^{x}C_{j} * {}^{y}C_{j} * j! * 2^j$

We can see, that $DP_{(i,odd)} = DP_{(i-1,odd)} * P(x)$ holds.

For a palindrome group with frequency x, Let's write two polynomials $P(x)$ such that $p_j = {}^{x}C_j * {}^{x-j}C_j * j!$ and $q_j = x * {}^{x-1}C_j * {}^{x-1-j}C_j * j!$

We can see that $DP_{(i,0)} = DP_{(i-1,0)} * P(x)$ and $DP_{i,1} = DP_{(i-1,0)} * Q(x) + DP_{(i-1,1)} * P(x)$

We can compute these products using Fast Fourier Transformation as nicely explained here 13

TIME COMPLEXITY

Using FFT, the time complexity is $O(N * M * log(M))$ per test case.

PROBLEM:

Given three integers N, M and K, Find the number of arrays A of length N such that the prefix sum array S of array A contains at least K elements divisible by M.

EXPLANATION

Let us solve a simpler problem first.

Given N, M and K, find the number of arrays of length N having each value in range $[0, M-1]$ such that exactly K values are 0.

Here, we want K values to be zero and the remaining $N-K$ values to be non-zero. For each non-zero value, we have exactly $M-1$ choices for that number. So we can select the non-zero values in $(M-1)^{N-K}$ ways.

But till now, we have only chosen the elements in order. They can appear at any of the $N-K$ positions among total N positions.

Assume $N=4$, $K=2$ and chosen non-zero values be $[1,2]$ in this order only. Then, there are six following ways to place them among array of size N.

```
0 0 1 2
0 1 0 2
0 1 2 0
1 0 0 2
1 0 2 0
1 2 0 0
```

It is easy to see, that it is equivalent to **Choosing** $(N-K)$ **positions** out of N positions and for each set of positions, choosing $(N-K)$ non-zero values, which can be done in $(M-1)^{N-K}$ ways, resulting in total ${}^{N}C_{N-K} * (M-1)^{N-K}$ ways to select array with exactly K zeroes.

Coming back to original problem, let us Consider prefix sum array modulo M, since We only care about values in prefix sum array modulo M, and $(a+b)\%M = (a\%M + b\%M)\%M$.

Let's call this modulo prefix sum array T. It can be seen, that all values in this array are in range $[0, M-1]$.

Let's assume $T_i = x$ where $0 \le x < M$. Now, what values can T_{i+1} take? We can see, that T_{i+1} can be $(x+y)\%M$ where y can take any value from 0 to $M-1$. It is easy to see that for each value of y, $(x+y)$ takes a different value, and y takes M distinct values, so T_{i+1} can take all distinct values irrespective of T_i.

Hence, we can consider all possible arrays T such that each value is within range $[0, M-1]$ and find a unique array A whose modulo prefix sum array is T. So, there is one-to-one mapping between array A and array T. So, the number of valid ways to choose A array is same as the number of valid ways to choose T array.

Hence, the original problem turns into choosing array T of length N such that each value is independent of each other and at least K values are 0 (Since only 0 in range $[0, M-1]$ is divisible by M). We can individually fix number of 0 to p for $K \le p \le N$ and count the number of arrays T with exactly p zeroes.

For computing binomial coefficients, it is better to compute factorials and their inverses in advance, and ${}^nC_r = \frac{n!}{r!(n-r)!}$ Details can be found here 129 .

TIME COMPLEXITY

Time complexity is $O(N)$ per test case.

PRE-REQUISITES:

Heavy Light Decomposition [12], Segment Tree [6] (It is expected that you know range product query on a segment tree), Basic Combinatorics [3], LCA [5]

PROBLEM:

Given a tree of N nodes (with each node having some value W_i), answer following queries-

- Update the value of some node to X
- Replace -1's in shortest path from u to v such that -
  - Starting from u to v, the series has the required relative order (i.e. strictly increasing, non-decreasing, strictly decreasing, non-increasing)
  - All numbers in path from u to v lie in range $[A, B]$

QUICK-EXPLANATION:

Key to AC- Realize that number of ways by which you can replace -1's is bounded by the non-negative W_i's (if existing) on both extremes of the path.

Realize
should
decrea
easiest
same l

Quote Share

Lets fo
increas

Let me call some value W_i as bounding value(s) if its not equal to -1, and if there is a group of zero or more -1 between it uptil next non-negative W_i if we follow a given direction. Now, notice that on a given path, the number of ways to replace -1 such that the sequence is increasing/strictly increasing depends on these bounding values. This is because no matter how large the values of A and B are, any value in middle of the path (and bounded by the bounding values) must lie between these 2 for the increasing/strictly increasing condition to hold good. We will use this in our implementation.

Use HLD to decompose the tree into chains, and make 4 segment trees as-

- One to answer the strictly increasing query if we go up from some node, say node u.
- One to answer the strictly increasing query if we go down from some node, say node u.
- One to answer the increasing query if we go up from some node, say node u.
- One to answer the increasing query if we go down from some node, say node u.

The directions are important so that we can reuse the same logic to answer the decreasing/strictly-decreasing queries by reversing the path.

We can store the results for each "group" of -1's in the segment tree (along with other data to keep the answer consistent, or obtain required intermediate values) and do a range product to answer each query in $O(log^2 N)$. Note that for the trailing and starting -1's you mught have to handle manually for easier implementation.

EXPLANATION:

The editorial will chiefly focus on the idea part. The implementation part of it, like any other HLD question, is a bit tedious - and the best way to learn it is to actually struggle through it and learn from other people's elegant solution. The best I can do is to give an idea on how the HLD is working, but the implementation is something one needs to do for himself.

I have divided the editorial into the following sections-

- How to reuse the logic for increasing queries to answer for decreasing queries.
- Significance of bounding values
- Combinatorics Involved
- Some notes on implementation

The answers of all "(Why?)" are in bonus section.

Regarding Queries

The first thing which makes a coder's eyeballs bulge on seeing the problem is that we have to answer 5 queries!! And each of them seem equally tedious.

And honestly, if one does not think and straight away goes head on to tackle the question just like that, then it will be an implementation nightmare for him. So lets see how to avoid it 🙂

Look at the strictly increasing query. It says that we need to replace each -1 in path from vertex u to vertex v such that all integers in this path are between $[A, B]$ and that the path is strictly increasing. Now, if the path from u to v is strictly increasing, this will also mean that the path from v to u must be strictly decreasing as well.

A similar argument holds good for the non-increasing and non-decreasing queries. That is, if the path from u to v is non-increasing, then the reverse path from v to u is non-decreasing!

This means that, if we can somehow account for the path and/or direction (which helps in accounting for path!), then there is a non-trivial chance to reuse the implementation for increasing and non-decreasing queries to answer the decreasing and non-increasing queries. How we achieve so, is an implementation aspect which we will integrate in the last section.

Significance of Bounding Values

For sake of clarity, lets consider the path from i to j having values W as following - $[W_i, -1, -1, -1, ..., -1, W_j]$, where W_i and W_j are not equal to -1. Notice that no matter what A and B limits are given by the query, the number of ways to fill -1 with some value such that the path is, say, strictly increasing, are fixed and dependent on values of W_i and W_j.

In other words, the number of ways depend on W_i and W_j and not on A and B for such cases, where all -1's are bounded by some non negative values of W_i and W_j . This is because none of the -1 in the range can be less than W_i or greater than W_j. (Obviously, if $W_i > W_j$ holds then the answer for our case of strictly increasing is 0 anyway). This means that the "true" bound for the group of -1's in between W_i to W_j is actually $[W_i, W_j]$ and not $[A, B]$.

Now, look at our path in the query from Node u to Node v. Now, see that you can look at the path as follows-

1. There MAY be some -1's in the beginning.
2. There MAY be some -1's at the end.
   - If number of $W_i > -1$ on the path is 0 or just 1, then above 2 cases cover it all.
3. If number of $W_i > -1$ is 2 or more, you will see that the path can be divided into "chunks" which I took an example of. That is, some non-negative W_i, then zero or more occurrences of -1 and then the next bounding value W_j. For those, the answer does not depend on A and B.

For cases pointed to by point 1 and 2, we can easily answer manually knowing the value of A, B and the number of occurrences of -1 which we have to fill. For the case pointed at by point 3, realize that since the answer depends on bounding W_i and W_j, we can pre-compute this at the beginning, and can update the results while handling updates. The important thing to note is, that we can avoid having to calculate this thing for all chunks again and again while answering the other 4 queries.

Now, we have $O(N)$ such chunks (Why?). However, doing range product and updating some node naively (eg- using arrays) will be $O(N)$ per query. Hence, we use a data structure like Segment Tree to do range product and updates in $O(logN)$

But how to calculate this number of ways? We will deal it in the Combinatorics section.

Combinatorics Involved

This section will deal with "How to select integers such that all numbers in the chunk are in range $[W_i, W_j]$

Lets say that the bounding values of our "chunk" is W_i and W_j. As we already saw that the solution to strictly decreasing and non-increasing queries can be calculated by reusing logic of increasing and non-decreasing, we will only discuss cases of increasing and non-decreasing.

For the 2 cases below, assume that for our "chunk" starting from node i to node j, our bounding values are W_i , W_j respectively. Also, assume that there are cnt occurrences of -1's to fill in the chunk.

- Increasing - This reduces to finding number of distinct elements in range (W_i, W_j) (Why?). Since both W_i and W_j are excluded, we have $W_j - W_i - 1$ options, and have to pick cnt out of those. Hence the answer is $\binom{W_j - W_i - 1}{cnt}$
- Non-Decreasing - What is the difference between above case and this one? In above case, repetitions were not allowed as it'd mean the sequence is not strictly increasing. However, in this case we can use repetitions. This reduces it to select integers in range $[W_i, W_j]$ with repetitions, which as proved here [2] to $\binom{W_j - W_i + cnt}{cnt}$

This deals with handling the chunks. But if you recall how we defined our path earlier, you'd recall that there is one more thing to touch. What about the -1's before our first chunk, or -1's after our last chunk, or what if there are no chunks at all!

- If there are no chunks, then it we have to fill all -1 with values in range $[A, B]$. Say there are cnt such values. You can easily find their formulas by substituting $W_i = A$ and $W_j = B$ in above formulas for non-decreasing query, and $W_i = A - 1$ and $W_j = B + 1$ for increasing query (as our formula for increasing excludes the end points!)
- Handing -1's before first chunk - Let the first chunk start with value K. Now, our problem reduces to filling all -1's with values in range $[A, K]$. Substitute $W_i = A$ and $W_j = K$ in above formulas for non decreasing query. For increasing query, $W_i = A - 1$ and $W_j = K$ (as we exclude both endpoints.)
- Handing -1 after the last chunk - Here, say the ending value of last chunk was K. Substitute $W_i = K$ and $W_j = B$ for non decreasing query, $W_i = K$ and $W_j = B + 1$ for the increasing query and find the answer 😃 .

This handles the combinatorics aspect of the problem! 😃 The next section highlights some implementation aspects, mainly taken from setter's solution.
Updates and Implementation Aspects

Lets first discuss the implementation aspects of the problem.

The very first thing is to use HLD to decompose our tree into chains, over which we can build a segment tree to answer range queries and handle updates. The implementation for HLD can in itself be lengthy, but it is something which cannot be helped 😕 .

The next task is to make segment tree over chains.

Notice that in our very first section, where we proved that the logic for increasing/non-decreasing queries can be used to answer the decreasing/non-increasing queries, we saw that direction mattered.

Like, having the sequence increasing in path from u to v meant having the sequence from path v to u to be decreasing. Or, allow me to state it like this-

"Having the sequence increasing from u to v is same as having the sequence increasing from u **UP** till $LCA(u, v)$, and then having it increasing from $LCA(u, v)$ **DOWN** to v".

Similarly, the reverse of this path can be defined as-

"Path **UP** from v to $LCA(u, v)$ and then from $LCA(u, v)$ **DOWN** to u".

Now, the direction where we are moving matters as it affects the boundary values of "chunks". Eg- going from i to j, if we encounter a chunk $[W_i, W_j]$ then going from j to i we will encounter the chunk $[W_j, W_i]$. Both of them will lead to different answers according to our combinatorics formula.

This means, that we should make one segment tree per direction for each type of query over a path/chain. Since we only need to implement for increasing and non-decreasing queries, the total number of segment trees we need is $2(directions) * 2(queries) = 4$. As summarized in quick explanation, they will be as follows-

- One to answer the strictly increasing query if we go up from some node, say node u.
- One to answer the strictly increasing query if we go down from some node, say node u.
- One to answer the increasing query if we go up from some node, say node u.
- One to answer the increasing query if we go down from some node, say node u.

Now, while making the segment tree, make sure you store sufficient parameters to help you answer the queries across various chains. Storing just the range product can be insufficient (Why?), although it really depends on your implementation.

Updates are pretty simple to think of. The thing to note is that you need to update it on all 4 segment trees we defined above (if your implementation is similar to the one discussed). The updates are point updates, which honestly do not leave a lot of things to be discussed. If the endpoints of a chunk are updated then its easy as only the contribution of the chunk is to be recalculated. Howver, be careful of updates where a -1 is updated to some non-negative value, or if some existing value becomes -1, as then new chunks may be formed.

Aside from that, the best thing to tackle implementation can only be to see what elegant tricks others have applied. The idea for the problem is very neat, however it demands command over implementation to give full score 🙂

SOLUTION

Setter

$Time\ Complexity = O(QLog^2N)$
$Space\ Complexity = O(N)$

CHEF VIJJU'S CORNER 😃

Why O(N) chunks?

Note on how we defined our "chunks". We defined our "chunks" as " Something beginning with non-negative W_i, followed by 0 or more -1's and then ending with another non negative value W_j."

The end points may belong to multiple chunks, and thats the case when some node with $W_i \geq 0$ has multiple children. In this case, each child leads to a unique path and hence a unique chunk. But the number of children is $O(N)$ and edges are also $O(N)$ and cycles are also not allowed, hence construction of the special test case where nodes are interconnected to lead to $O(N^2)$ chunks (and hence requirement of $O(N^2)$ paths intersection at only end points to cover the entire tree) is not possible.

Similarly you can argue about the values in middle of chunks (i.e. -1)

Combinatorics: Why the case of increasing became number of ways to select integers from range(Informal Intuition)

The thing to ask is that, when will 2 cases be different?

In our $\binom{n}{r}$ formulas, the order does NOT matter. Now, what the does our $\binom{n}{r}$ formula represent? It represents that irrespective of order, there are these many ways to select r things from n objects. Since order does not matter, I can simply put all selected numbers in increasing order - that will be one way.

Now, look at the increasing sequence. No other order or permutation can be increasing, because swapping any value will lead to the sequence to be no longer increasing. Hence, there is only 1 possible order/way-of-putting-the-numbers.

Now, two increasing sequences can be different iff at some index i, $seq1[i] \neq seq2[i]$. But if that is the case, this means that the numbers selected are different. Had it been the case that some permutation of same selected numbers would have satisfied the conditions, then our formula would go wrong. But we see that only 1 permutation of selected numbers can satisfy the cases, and two sequences can be different only if number selected at some positions are different.

Hence the number of ways to replace the -1 in the chunk increasing order is equal to the number of ways to chose cnt numbers in range $[W_i, W_j]$.

Why storing just the range product can be insufficient

This is a general perspective and may/may-not be applicable to the current problem. As I said, implementations vary.

The thing to note is that, HLD decomposes the tree into disjoint paths such that you can reach root from any vertex in $O(LogN)$ path changes.

Over each (heavy) path/chain we build a segment tree. You must store sufficient parameters such that all cases are handled.

Say I am going from u to $LCA(u, v)$ which are on one path. Lets say that $W_{LCA(u,v)} = -1$, and that its W_j would be found somewhere down when going from $LCA(u, v)$ to v.

Lets say if I store the number of -1 in prefix/suffix of the chain, I might be able to check if $W_{LCA(u,v)} = -1$ or not and handle that one case manually. But if all that I stored is the answer, then I might get wrong answers when changing across chains.

Setter's Notes

We can build an HLD there. Then we can split every path into $O(log(N))$ paths. You can note that only values with A_i not equal to -1 matter.
So we can check whether these values satisfy the non-decreasing condition. If no then the answer is 0.
Now for every A_i not equal to -1 let's maintain the number of ways to change -1 till the next A_i not equal to -1 on its heavy path(This can be done easily with some binomial coefficients and if we store set of A_i not equal to -1 on every heavy path) And store this values in

segment tree. For -1 these values should be equal to 1. Now roughly speaking we can query product on the range in segment tree. And for most left and rightest A_i on the path we can find the number of ways to change values of -1 on it and on the previous heavy path by hand again with some binomial coefficients.

$Update : O(log(N))$
$Query : O(log^2(N))$

- Make sure to give this 22 unofficial editorial a read as well as it gives some more perspective on segment tree implementation. With respect to that editorial, answer the following-
  - Describe the use of each parameter stored in his segment tree. Don't tell what it does, tell why it is needed.
  - Describe the parent child relationship of his segment tree
  - Leave a like at his editorial so he no longer has to ask for sympathy votes 😃

Related Problems

- Queries on a Tree 2
- Queries on Tree Again 2
- Black and White Tree 1
- Tree 4

2

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
RIP2000 - Editorial editorial editorial start97	0
MAXSUMOPS - Editorial editorial editorial start112	0
SWAPUNITE - Editorial editorial binary-search editorial prefix-sum start124	5
PREP41 Editorial editorial	0

There are 39 unread and 7 new topics remaining, or browse other topics in editorial

PREREQUISITES:

Basic Combinatorics, Enumerating/Iterating over combinations

Hint: Consider all possible combinations of choosing oranges and see what you can do with them.

EXPLANATION

subtask1: The weight for each orange is given to be 1.

To pick the oranges such that we get a maximum weight purchase is the same as picking the most number of oranges. This can solved by a simple greedy approach. Just sort the array of costs in increasing order. Traverse the array from left to right and see how many oranges can you buy using the k rubles that you have. The answer will be the number of oranges that you can get.

subtask2: n = 5

Given five oranges you can choose your oranges from one of the following 31 ways (Consider a string of “abcde” of length 5, each character can be 0 or 1. If ith character is 1 we take ith orange otherwise we don’t.)

C(5, 1) = 5 possible combinations of choosing exactly one orange.

[“00001”, “00010”, “00100”, “01000”, “10000”]

C(5, 2) = 10 possible combinations of choosing exactly two oranges.

[“00011”, “00101”, “00110”, “01001”, “01010”, “01100”, “10001”, “10010”, “10100”, “11000”]

C(5, 3) = 10 possible combinations of choosing exactly three oranges.

[“00111”, “01011”, “01101”, “01110”, “10011”, “10101”, “10110”, “11001”, “11010”, “11100”]

C(5, 4) = 10 possible combinations of choosing exactly four oranges.

[“01111”, “10111”, “11011”, “11101”, “11110”]

C(5, 5) = 10 possible combinations of choosing exactly five oranges.

[“11111”]

Just go through all possible combinations and among the combinations which have a cost less than equal to k rubles pick the one yielding the maximum weight. You can hardcode the previous combinations but computing them by hand may be very tedious so it is not recommended unless you get it is more clever way like copying them from some other source on the internet. We will now see a way of doing this using code. Consider the following code.

```
for (i = 1; i <= 5; i++) {
    //take ith orange
    for (j = i+1; j <= 5; j++) {
        //take ith, jth orange
    }
}
```

In the outer for loop you can select exactly one of the oranges 1,2,3,4 and 5. In the inner for loop you can select all possible ways of choosing 2 oranges (1, 2), (1, 3), (1, 4), (1,5), (2, 3), (2,4), (2,5), (3,4), (3,5),(5,5). Using more for loops you can get a way to select all possible ways of 3 oranges, 4 oranges, 5 oranges. Every time you select a subset of oranges just see if they don’t cost more than k rubles and among all such possible subsets pick the one with the maximum weight. Refer to editorialist’s solution for an implementation of this.

subtask3: (solves subtask2 and subtask1 as well)

We will select all possible combinations of choosing from the n oranges and among whichever combinations cost us no more than k rubles we pick the one with the maximum weight. The number of different ways of choosing oranges from the given oranges is same as the number of subsets of a set which is 2^n (this includes a way of selecting no orange as well). Since n is at max 10 we are looking at a maximum of 2^10 = 1024 combinations, so the problem is easily solvable this way. You can solve this in multiple ways.

Method 1:

You can iterate through all the subsets of the n oranges by using simple bit level operations. Read through the tutorials at http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation 13 and http://www.codechef.com/wiki/tutorial-bitwise-operations 11 . You can see the tester's code for an implementation of this mechanism. This is the usual expected and recommended way of doing this kind of a thing in contests.

Method 2:

You could precompute all possible subsets in the following way. We will use string representation used in subtask2. We use a string S of length n to represent a subset, if ith position is 1 we select ith orange otherwise we don't.

Have a two dimensional array combinations11 13 (Since n <= 10)

combinations[i][j] holds a list of all strings denoting the ways of selecting j oranges from a total of i oranges. (j can't be greater than i)

We will initialize combinations[0][0] with a list having just the empty string.
We will initialize combinations[i][0] with a list having just a string of length i made up of all 0's.

We can compute combinations[i][j] recursively as follows.
We want to generate all i length strings having exactly j bits set to 1. We can set 1st bit to either 1 or 0. In the first case we can add 1 in front of the list of all strings of combinations[i-1][j-1]. In the second case we can add 0 in front of all the strings of combinations[i-1][j] (only if i-1 >= j otherwise this would lead to absurdity). These are the only possible strings of length i and having j bits set. So we can add all these strings to a new list and set them to combinations[i][j].

We can compute the whole combinations array either recursively or iteratively. Refer to editorialist's solution to look at a way of generating this iteratively.

Once we have this generated for a given n . We look at all the strings in the lists combinations[n][1], combinations[n][2], ……, combinations[n][n] and for each string we see the oranges that are to be selected and compute our maximum weight needed.

Method 3:
Since n can be at max 10 we can simply use 10 for loops just like the 5 for loops used in subtask2. Given the nature of for loops conditions if there is anything to be selected it will be executed otherwise the for loop condition fails in the initial check it self, so the previous method just works seamlessly in this case. Refer to the method subtask3_forloops in editorialist's solution. This is a bit ugly solution and has a chance of some typos getting in when dealing with so many loop variables. If you are a beginner this might seem the simplest of all three solutions but on greater practice you will realize that the method 1 is in fact the simplest solution to implement in this case.

If you are using a language like python which has in built functions to generate combinations your task becomes very easy. See the editorials python solution. Note that currently python is not a supported language in either IOI or ICPC, but it is supported on all major online judges.

Note: This problem looks like a standard knapsack but the DP solution for knapsack for this problem would get a Time Limit Exceeded or Memory Limit Exceeded message because of cost being as big as 10^8.

PREREQUISITES:

Combinatorics and Factorials. Modular Arithmetic.

PROBLEM:

Given an array A of size N, Count the number of ways we can select a subset from A such that the median of selected subset is also present as an element in the subset.

SUPER QUICK EXPLANATION

- Every subset of odd size has its median present in the subset, so, we can directly add 2^{N-1} to answer.
- For even size subset, The subset is good, if and only if middle two elements are equal.
- We can fix the left middle element, and for every possible even size of the subset, say $2 * X$, try to

EXPLANATION

This problem is probably the best example of how we can use combinatorics to optimize our solution from $O(N^3)$ to $O(N)$ (except sorting).

First of all, let us formulate the definition of median.

If the size of the selected subset is odd, the median is just the middle element of subset after sorting. Since the middle element is present in the subset, all subsets of odd size are valid. It can be easily proven that there are 2^{N-1} such subsets. So, we can directly count these and move toward even size subsets.

If the size of the selected subset is even, the median is defined as the average of two middlemost elements after sorting. Now, say we have two middle elements x and y, with condition $x \leq y$. Let $z = (x + y)/2$ be the median of sequence. If we write $y = x + d, d \geq 0$, we can see, $z = x + d/2$ and also, $z = y - d/2$. This way, we can see, that the median of a sequence can never be smaller than x and greater than y. So, For z to be present in subset, we need either $z = x$ or $z = y$. But, this would imply $d = 0$, Hence leading to the conclusion that

For an even size subset to be valid, the two middlemost elements should be equal. This forms the crux of our solution, and now, we need to count the number of even sized subsets with equal middle elements.

After all this, there are a number of approaches to solving this problem, all of which required us to sort the array A first.

Let us consider a $O(N^3)$ solution first.

Iterate over every pair of equal elements (i, j) such that $A_i = A_j$ and iterate over the size $2 * X$ of subset from $X = 1$ to N. The number of ways to make the subset of size X with two fixed middle elements is just the product of the number of ways we can select $X - 1$ elements from $[1, i - 1]$ and $X - 1$ elements from $[j + 1, N]$.

This solution requires to iterate over every pair (i, j) which takes $O(N^2)$ time and $O(N)$ time per pair, leading to Overall time complexity $O(N^3)$ which shall pass only the first subtask.

For a faster implementation, We will see two implementations, one from setter, and another interesting solution which we can further optimize to $O(N)$ except for sorting.

Setter's Implementation

First of all, see what we were doing in the previous solution.

We were fixing two equal elements and tried to count the number of ways we can make subsets of all sizes. Now, We shall fix only the **Left Middle Element** (Or Right one, whichever implementation you prefer).

Suppose we fixed the ith element as the left middle element. Now, We will iterate over all sizes $2 * X$ and try to include $X - 1$ elements from $[1, i - 1]$ and X elements from $[i + 1, N]$. We need the right middle element to be same as the left middle element. So, When choosing X elements from $[i + 1, N]$, we need at least one occurrence of $A[i]$. This is same as subtracting all the ways to select X elements in the range $[i + 1, N]$ which do not have $A[i]$ at all. Suppose Number of occurrence of $A[i]$ in range $[i + 1, N]$ is f, then we can count the number of ways to select X elements from range $[i + 1, N]$ such that it contains at least one occurrence of $A[i]$ as

T = Number of ways to select X elements out of $N - i$ elements less Number of ways to select X elements out of $N - i - f$ elements.

We can select $X - 1$ elements from $[1, i - 1]$ in suppose U ways. Then, Number of ways we can have good subsets with $A[i]$ as the left middle element is $U * T$. Summation of this product for all sizes for all elements gives us the number of good subsets of even size. We can add to it, the number of good odd sized subsets and print the answer.

Alternative Implementation.

For this solution, We will not count good subsets, but subtract bad subsets from total subsets. The total number of subsets is 2^N out of which one is empty. Empty subset doesn't contain its median, hence a bad subset. So left with $2^N - 1$ subsets.

In this solution, we will count the number of subsets which have the different value of middle elements. For this solution, we iterate over all distinct elements and try to the number of ways to select subsets of all even sizes.

In this solution too, we fix the left middle element and try to count the number of bad subsets.

Suppose we have cnt elements smaller than $A[i]$ and There are total f occurrences of $A[i]$ and we have to make a subset of size $2 * X$, Then, we need to select X elements from $cnt + f$ such that it contains atleast 1 occurrence of $A[i]$ and selecting X elements from Elements strictly greater than $A[i]$, THere are $N - cnt - f$ such elements. The required number of ways is the product of both. We can find the summation of this product over all even sizes of subsets for all distinct values.

This solution also has time complexity $O(N^2)$, thus fits the time limit easily.

Hint to O(N) Solution. Note: This optimization is not exactly necessary.

We rely on the same idea as the alternative implementation. We still iterate over all distinct values, but now, we will find a closed form to count bad subsets of all sizes. But the thing is, we try to find a closed form for all sizes.

We see, the summation is like ${}^nC_x * {}^mC_x$ for all sizes x. Can you find the closed form? Seems like many users actually did, and submitted a Linear time solution during the contest itself.

For those who did not reduce, this is left as an exercise for them.

Looking for something helpful? See Vandermonde's identity here 27 .

Computing Number of ways to select X elements out of N elements

The thing is, we need to calculate N select X efficiently. We know from the definition of Combinations 3 that we can represent them in form of factorials. We can precompute Factorials and their inverses for the division. Then, ${}^NC_R = N! * inv(R!) * inv((N - R)!)$.

For details, refer this excellent post here 15 .

Time Complexity

Time complexity is $O(N^2)$ per test case for the intended solution.

PRE-REQUISITES:

Basic Combinatorics, Lucas Theorem

PROBLEM:

Find the number of graphs possible N vertices and M edges.

All vertices and edges must be used for a graph and at least one graph will be possible for every input.

QUICK EXPLANATION:

The answer for each input is given by :- $^{n}C_{2}C_{m}$ % 2713
this can be easily computed using lucas theorem, preprocess some values for faster computation

Explanation:

The number of different simple undirected graphs with n vertices and m edges is $^{n}C_{2}C_{m}$.

Let us prove this:

No. of ways to choose any two vertices : $^{n}C_{2}$ as it is undirected so arrangement dosen't matter

This can be thought of as having $^{n}C_{2}$ edges to choose from

Now, we have to choose 'm' edges from $^{n}C_{2}$ possible edges, which is given by $^{n}C_{2}C_{m}$

Alternative proof:

Let us consider the adjacency matrix of the graph

Total cells = n^2

As the graph is simple, there are no self loops, so we eliminate the diagonal cells.

So, now we're left with $n^2 - n$ cells

Since the graph is undirected, $mat[i][j] = mat[j][i]$, so we'll only have to choose from one half(bottom or top) as other half cell will be automatically filled

Therefore, cells to fill = $(n^2 - n)/2$

Since we have 'm' edges, we have to fill 'm' cells among these cells

So ways to fill it is $^{(n^2-n)/2}C_{m}$

Okay, so now we know that we have to find $^{n}C_{2}C_{m}$ % 2713,

but if we go about finding $^{n}C_{2}C_{m}$ directly and then do modulus, it will take too much time(time complexity will be huge) for the given constraints. see here for more information on such topics

If we analyze the modulus value, we come to know that it is a prime and a small one
So the best way to find our answer is to use Lucas Theorem(Link 1 , Link 2)

Basically you need to convert the numbers in base 2713 and then multiply the successive combinations

If we preprocess the combinations, time complexity for each test case would be: $O(log_p n)$

Pitfalls:
The assumption that ${}^{n}C_2 C_m \bmod 2713$ can be written as ${}^{(nC2 \bmod 2713)}C_{(m \bmod 2713)} \bmod 2713$ is incorrect especially for large values

Some test cases with answers:

Input:

```
13
1000000000 11543263
1000000000 153126357391543263
37889434 334278564736253
99335555 417333344
1837 741
8242 2914
24191 19337
9219 3957
10 44
16347 5200
8385 8335
21132 7180
156600 1378774
```

Output:

```
0
0
471
0
2340
1838
874
2605
45
0
1808
0
1735
```

Time Complexity: $O(p^2 + log_p n)$

Space Complexity: $O(p^2)$