Prerequisites

Combinatorics

Problem

Given a string S and an integer K. Find the number of strings, T, having the same length as S such that T is palindrome and the hamming distance between S and T is at most K.

Some Hints/Tricks

Try to think about how changing an index in left part or sting S, forces us to operate on the corresponding right index to maintain the palindromic property satisfied in final string. Try to find a combinatorial formula for it. Precompute some sums (or formulas) to get an overall complexity of $O(n)$.

Explanation

The question can be reframed as “Find the number of strings which can be formed by changing at most K characters in S, so that resulting string becomes a palindrome.” Assume, 0-based indexing in the editorial.

The strings T that we need to form should be palindrome. This restricts how we modify some characters from string S. Thus, we define 2 terms here:

1. **Fixed:** The set of those indices, $i(< \frac{n}{2})$, such that $S[i] == S[n-1-i]$.
2. **nonFixed:** The set of those indices, $i(< \frac{n}{2})$, such that $S[i] \neq S[n-1-i]$.

Based on these terms, we can clearly see that the all possible strings, with given counts of “Fixed” and “nonFixed” indices, will give the same answer. This is because the answer depends on whether we chose to perform an operation on given index or not and this may force us to perform some operation on another index so that final string remains palindrome. These operations don’t depend on the characters present at that location in the string.

Thus, to calculate the complete answer, we need to evaluate 2 functions, namely:

1. $f1(Fixed, k)$ = returns the number of ways to change “Fixed” indices so that palindrome condition is satisfied at those indices, with the condition that **maximum of** k changes can be made.
2. $f2(nonFixed, k)$ = returns the number of ways to change “nonFixed” indices so that palindrome condition is satisfied at those indices, with the condition that **exactly** k changes can be made.

From the definition of above 2 functions, it is easy to see that the final answer is as follows:

$$F(S,k) = \sum_{i=0}^{i=k} f1(Fixed, k-i) * f2(nonFixed, i)$$

Some simplification, before we proceed (assuming we can implement the above functions efficiently). What happens when the length of the given string is odd. The middle character will be not be treated as “Fixed” or “nonFixed”. But, it is clear that we can change this character without affecting the palindromic property in final string. To fix this, we can either change that character or not. Thus, final answer, in this case, would be as follows:

$$ans = F(S,k) + 25 * F(S,k-1).$$

Now, let us come back to the evaluation of functions $f1(Fixed, k)$ and $f2(nonFixed, k)$ defined before. These are given by following formulas :

$f1(Fixed, k) = \sum_{x=0}^{x=floor(k/2)} (^{Fixed}_{x})25^{x}$

$f2(nonFixed, x + 2 * (nonFixed - x) <= k) = (^{nonFixed}_{x}) * 2^{x} * 24^{nonFixed-x}$, provided that ($x >= 0$ and $x <= nonFixed$)

The reasoning is as follows:

1. In “fixed” case, the condition of the palindrome is already satisfied. So, if we change a character at fixed index, (say i), we need to change it at another location, (namely $(n - i - 1)$). Thus, we every time need to make at least 2 changes in S. Thus, we select the number of indices we want to make changes in and then make changes in them. The number of ways to chose the indices is $(^{Fixed}_{x})$ and the number of ways is 25^{x} as there are 25 different possible characters to be replaced at each location.
2. In “non_fixed” case, the condition of the palindrome is not satisfied. So, we can either change both the indices, i and $(n - i - 1)$, or change only one of them. **But change has to be applied to ensure final string has the palindrome condition satisfied**. If we change both of them, then both of them must be same and there are remaining 24 characters to chose from. If we change only one of them, then we have only 2 choices, either change $S[i]$ and make it equal to $S[n - i - 1]$ and vice-versa.

Implementing the above full idea naively will take $O(N^2)$ time as there will be double summations. The trick is to pre-compute the function $f1(Fixed, k)$ and compute the final answer, $F(S, k)$ in such a way that function $f2(nonFixed, k)$ is computed on the fly.

To compute $f1(Fixed, k)$ for all values of x, such that $0 <= x <= k$, we can make one observation. The value of $f1(Fixed, k)$ will be same as $f1(fixed, k - 1)$, if k is odd. Otherwise, the only additional contribution is of last term i.e. $(^{Fixed}_{k/2})25^{k/2}$.

To get more idea about the implementation, refer to editorialist solution. In case of any doubt’s, feel free to comment below.

Time Complexity

$O(N)$, per test case. Assuming all precomputations are also done in $O(\text{MAX VALUE})$.

PREREQUISITES:

Probability, Expectation, Dynamic Programming and Combinatorics.

PROBLEM:

Given a tree with N nodes numbered through 1 to N, each having a color, initially colored white. Find the expected number of steps the following process will last.

In each step, choose an unordered pair (a, b) of vertices at random, if there's any node colored black on the path between a and b, terminate the process immediately. Otherwise, color all vertices on the path between a and b black and move to the next step.

SUPER QUICK EXPLANATION

- Reduce Expectation to by writing $E(X) = P(X \geq 1) + P(X \geq 2) + \dots P(X \geq N)$, where $P(X \geq T)$ means Probability of performing at least T steps. We can see, that If NumPath is same as Number of ways to choose at least Y paths, then $P(X \geq Y)$ is $numPath * K!/((N * (N + 1))/2)^K$
- We can maintain a DP table for every node, counting the number of ways to make X paths in the subtree of the node, with another state, telling whether the current path can be connected to paths coming from the parent of current node.
- To calculate this DP from child to its parent, we need to used another DP table, dp2[X][Y][ST], which counts number of ways to make Y paths using first X nodes such that $ST = 0$ means current node is not used at all, $ST = 1$ means the Current node is included in a path which has one end at current node.

EXPLANATION

So, Time for another combinatorics problem.

First of all, If X is a random variable, Expection is calculated as $1 * P(X == 1) + 2 * P(X == 2) + 3 * P(X == 3) \dots N * P(X == N)$. But, we can also rewrite it as

$$E(X) = P(X \geq 1) + P(X \geq 2) + P(X \geq 3) \cdots + P(X \geq N).$$

Now, see what $P(X \geq K)$ means. It means **The number of ways to select at least K non-intersecting paths, in a given order.** Now, since Order of selecting path is irrelevant as all paths do not intersect, We can write it as $K! * NumWays$ where $NumWays$ is the number of ways to select at least K paths. Total number of possible paths selected (including intersecting ones) is $((N * (N + 1)/2)^K$, Hence, $P(X \geq K) = NumWays * K!/(N * (N + 1)/2)^K$.

The problems get reduced to calculate NumWays, and for that, we resort to Dynamic Programming.

First thing, We can have at most $sub[u]$ non-intersecting paths in the subtree of u, where $sub[u]$ is the number of nodes in the subtree of u. The reason is, that suppose, each path cover exactly one node. Even then, the number of Non-intersecting paths is $sub[u]$ and it can be seen, that we can only decrease Number of non-intersecting paths.

Now, we will calculate for each node u, Number of Ways to select X nodes for $1 \leq X \leq SZ[u]$. To calculate this, we also need information whether we can connect the current node with its parent in a path.

There are two types of paths in the subtree of node u which include u.

- Paths coming from the parent cannot be connected to current node/subtree of the current node.
- Paths coming from the parent can be connected to current node/subtree of current node. Call it **Special Path**

See, that special type can also be extended toward the parent of u, thus, needs to be handled separately. Also note that the path of the second type may choose to end at the current node, hence, will always be included automatically in paths of type 1.

Now, suppose we have it calculated for every child v of u, the number of ways to select at least k paths in the subtree of v for $1 \leq k \leq sub[v]$. We need to compute the same for the current node.

For computing this, we can use another DP for each node, call it dp2[childUsed][NumPaths][0/1/2] which calculate the number of ways we can select NumPaths in children of first childNodes children. The last state represents the number of special paths. We can assume Path of type 1 to be 2 special paths, to simplify our implementation.

We can see, that in no case can a node have move than 2 special paths, since Any path is the shortest distance between two nodes, and such path cannot visit more than 2 subtrees of any node.

Basically, we can fill the whole table using the recurrence.

$$dp2[child+1][mxPaths+curPaths][st1+st2] += dp2[childUsed][mxPaths][st1] * dp[child][curPaths][st2] \text{ if } st1+st2 \leq 2$$

Let us try all combinations of $(st1, st2)$ for understanding.

- $(0,0)$ This combination counts the number of ways of selecting paths, that child has no special path, and earlier child also do not have any special paths.
- $(0,1)$ This combination counts the number of ways of selecting paths, that child has a special path, while the earlier child had no special paths
- $(1,0)$ This combination counts the number of ways of selecting paths, that child has no special path, and earlier child have one special path.
- $(1,1)$ This combination counts the number of ways of selecting paths, that child has a special path, and earlier child have one special path.
- $(2,0)$ This combination counts the number of ways of selecting paths, that child has no special path, and earlier child have two special paths.

Combination $(2,1)$ cannot be considered because we cannot have more than 2 special paths coming to a node. Also, For any child, only 1 special path can be extended to its parent. Because if there are two special paths which are extended to parent, it cannot be the part of any shortest path between two vertices.

The reason is, that if two special paths are there, The only shortest path, the current node can be a part of, it the one starting somewhere in the subtree of one child of the current node and ending in a different child's subtree.

Hence, Only above five combinations need to be considered.

Now, we can see, that by the Fundamental principle of Counting, these are dependent events, and hence, are multiplied.

This way, we will obtain, after computing internal DP, the number of ways to select K paths out of all children of the current node such that the number of special paths up to children level is T ($T \in [0,2]$).

Now, we need to consider cases as to update the original DP using the values of this internal DP.

The thing is, that we cannot have two special paths extend toward its parent. So, we need to carefully count Number of ways to select paths for each count of special paths which can move forward.

We can see, that we can choose not to extend any path, extend one path to parent as well as make a path at the current node. You need to handle these cases to get Required Number of ways for the current node.

For implementation, please refer the tester's solution below.

Related Problem

A very recent problem Standard Tree Task from August Cook-Off which uses the precisely same type of Dynamic Programming, which you may refer here [3] .

Time Complexity

Time complexity is $O(N^2)$ per test. Seemingly, Each DFS takes $O(N^2)$ time, but due to Number of nodes being bound to N, Overall Time Complexity is amortized to $O(N^2)$.

PREREQUISITES:

Greedy, Understanding of Activity Selection problem, Dynamic Programming, and Combinatorics.

PROBLEM:

Given four integers N, M, K and MOD, determine the number of ways to select M intervals (possibly intersecting) in the range $[1, N] \bmod MOD$ such that the maximum size of disjoint intervals set we can select is K.

SUPER QUICK EXPLANATION

- For calculating answer for (N, M, K), we need to know answers for tuples like $(N - A, M - B, K - C)$ for valid values of A, B and C, leading to overlapping subproblems as well as optimal substructure property, hinting toward Dynamic Programming Solution.
- Let us represent $dp[n][m][k]$ as the number of ways to select m intervals in the range $[1, n]$ such that the maximum size of disjoint intervals set we can select is k AND the last selected interval ends at N.
- For transitions, we need to use combinatorics to move from current position to next Position, Counting the number of intervals which start after current position but before or on next Position, and end on Positions on or after nextPosition.
- The answer for our problem will be $\sum dp[i][M][K] \forall i \in [1, N]$.

EXPLANATION

Let us consider the reverse of this problem first, the problem to select the maximum number of disjoint intervals out of given intervals.

We can see, that the idea was to sort the given intervals in the non-decreasing order of right end of intervals and try to select an interval whenever possible. It will be possible to select the next interval if the right end of the previously selected interval is to the right of the left end of the current interval. Since intervals are ordered by right end, it guarantees to select the interval with leftmost right end whenever possible, leading to the selection of maximum disjoint intervals.

For more details, refer this [1] .

Coming back problem, Let us define the concept canonical sequence of disjoint intervals. See, the way we selected intervals, we can see, that the intervals have been selected in the increasing order of right end.

We can see, that there might exist multiple sets of intervals having the same size, or just the same set of intervals selected in different order.

So, to tackle this, Let us define the concept of the canonical sequence of this set of intervals as the right ends of selected intervals. Since we can see, that $r_1 < r_2 < r_3 \cdots < r_x$. The best thing of this sequence is, that it remains the same for a given set of intervals since we see, there is no randomness involved in the greedy solution we use to find the solution. This means, that if we select the intervals in increasing order of right end, we don't need to worry about multiple sets and orderings.

Suppose we have X intervals sorted by the right end in increasing order, and assuming we can select Y disjoint intervals at max. Now, suppose we start deleting intervals from right to left one by one, till the maximum number of disjoint intervals we can select is $Y - 1$. This way, calculating Y in that case becomes a smaller sub-problem, that is, to calculate $Y - 1$ from a smaller set of intervals, and add 1.

Since we see that we need to solve the same subproblems multiple times, we resort to our old friend, Dynamic Programming.

Let us define $dp[n][m][k]$ as the number of ways to select m intervals in range $[1, N]$ such that the length of the canonical sequence is k, and the last element of canonical sequence is n, that is, the last interval included in the canonical sequence ends at n.

We can see, that due to condition “the last interval included in canonical sequence ends at n”, final answer is the summation $\sum dp[i][m][k] \forall i \in [1, N]$.

Let us move toward State transitions. For convenience, let us call intervals present in canonical sequence as special.

See, suppose we have calculated the number of ways for dp[pos][selected][special]. We will try to count ways to add one more special interval for all combinations of (Right end of next interval, Number of Additional intervals selected), writing it as (nextPos, more).

See, the criteria for the next interval to be included in the canonical sequence is, that the interval should start after pos. Also, the start point has to be before or at nextPos since the right end of this interval is nextPos, according to our definition of DP state. We want to select a total of $more$ intervals.

But, Let us count the number of intervals which start in the range $(x, y]$ and ends in the range $(y, z]$. Since for every start point, it is possible to choose any right end in a given range. Number of start positions is $(y - x)$ while number of ending positions is $(z - y)$, Leading to a total of $(y - x) * (z - y)$ intervals. Suppose, it is also possible to select y as the right end, then the number of intervals would become $(y - x) * (z - y + 1)$. Understanding this will be helpful.

Now, See, need to select $more$ intervals such that all intervals have a common point so that exactly one interval gets selected in canonical sequence. And that common point we have is pos since our greedy algorithm chooses the interval with the leftmost right end. The number of ways to select $more$ intervals out of total $A = (nextPos - pos) * (N - nextPos + 1)$ intervals is given by $C(A, more)$.

But, it will also include the case where all intervals end after $nextPos$. It can be easily seen that number of such intervals starting after pos and before or at nextPos, and ending after nextPos is $B = (nextPos - pos) * (N - nextPos)$, out of which we have counted the number of ways to select $more$ intervals. This is given by $C(B, more)$.

Hence, Number of ways to select $more$ intervals which start after pos but before or at $nextPos$, and end at or after $nextPos$, while having at least one interval ending at nextPos, is $C(A, more) - C(B, more)$.

Now, After understanding all this, the problem left is to take the summation

$$dp[nxtPos][total + more][selected + 1] = \sum dp[pos][total][selected] * (C(A, more) - C(B, more)) \forall pos \leq nxtPos$$

Implementation of this problem is really simple. We can precompute the whole Combination table which will allow constant time lookup and is usually recommended because it uses only addition and modulus operation due to recurrence $C(N, R) = C(N - 1, R) + C(N - 1, R - 1)$.

Read more about calculating $C(N, R) \bmod P$ here [4] .

Time Complexity

Time complexity is $O(N^5)$ per test case theoretically, but it runs much faster in practice.

Pre-Requisites :

Binomial Coefficients

Problem :

You are given $2 \times N$ grid. Now, you need to color K cells in this grid, such that no two colored cells are adjacent. Find the number of ways to do so.

Explanation :

First of all, we can obviously note that no two bricks will be placed in the same column. So, let's just forget this scenario. Let us try and find the number of ways to place k bricks so that no two adjacent columns contain a brick in the same row.

Sub-task #1 :

Here, N is only up to 1000. So, we can maybe write some kind of $O(N \cdot K)$ dp, to pass this subtask. Consider the following dp :

$dp[i][j]$ = Number of ways to place j bricks among the first i columns in a valid way, such that the i^{th} column contains the last brick placed.

So, $dp[i][j] = \sum_{k=0}^{i-2} dp[k][j-1] \cdot 2 + dp[i-1][j-1]$. If any column $\leq i-2$ contains the last placed brick, then we can place a new brick in the i^{th} column in any of the two rows, and if the $(i-1)^{th}$ row contains the last placed brick, then we can place a brick in the i^{th} column only in the row not equal to the row of the $(i-1)^{th}$ column's brick.

But this dp is still $O(N^2 \cdot K)$. We can optimize this dp, coming to $dp[i][j] = dp[i-1][j-1] + dp[i-2][j-1] + dp[i-1][j]$, by using prefix sum optimization. Just compare the terms difference in the terms added by $dp[i][j]$ and $dp[i-1][j]$ tp get to this faster recurrence. Pre-compute this dp for all $1 \leq n, k \leq 1000$, and we can for fixed N, K sum up all values $dp[i][K], K\ i \leq N$.

Time Complexity $O(maxn \cdot maxk + \sum N_i)$, where $maxn = maxk = 1000$

Sub-task #2 :

Here , N is up to 10^9. Now, we need to make more observations. Let us consider the state each column can be in. Let the state $\begin{bmatrix}1\\0\end{bmatrix}$ denote we place a brick in the first row of this column, and nothing in the second row, Similarly, let the state $\begin{bmatrix}0\\1\end{bmatrix}$ denote we place a brick in the second row and nothing in the first row, and let the state containing all zeros denote we place nothing in this row. Now, we need exactly K columns to be in state $1/2$ and the rest in state 3 .

Also, in the final arrangement we need that no two columns that are in state $1/2$ to be adjacent to a column in the same state. Let's rename the states, consider state $1 = a$, state $2 = b$ and state $3 = c$. We can now solve an equavalent problem : Find the number of N length strings of character-set a, b, c consisting of exactly K $a/b's$, such that no two $a's$ are adjacent and no two $b's$ are adjacent.

Now, consider the number of i length strings consisting of only $a/b's$ and the number of positions j where a character equals its previous character. We can store this in $f(i,j)$,

$f(1,0) = a, b = 2$

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

We can pre-compute this table in $O(K^2)$ time. Now, for a fixed N, K, the answer is :

$\sum_{j=0}^{k-1} f[k][j] \cdot \binom{n-j}{k}$. This can be derived from the stars and bars method. We have finished placing the $a's$ and $b's$, k of them and we have $n-k$ $c's$ left. Now, when there are exactly j positions when a character equals its previous character, then we compulsorily need to place a c between them. Let's try and place $c's$ in between any 2 adjacent positions in the string of length k, or in the beginning/end.

So, this is : we have to represent the number $n-k$ as the sum of $k+1$ numbers, but j numbers have to be at least 1. So, by the stars and bars method, this is :

$\binom{k+1+n-k-j-1}{k+1-1} = \binom{n-j}{k}$.

Now, $\binom{i}{j} = \frac{i}{j! \cdot (i-j)!}$, We can calculate this by first taking product of all integers $x, x \leq i, x > (i-j)$, there can be at max k such numbers, and then multiplying by $(j!)^{-1}$, which we can pre-compute initially in $O(K) time$

It is easy to prove that $f(i,j)$ is actually a binomial coefficient multiplied by some constant. So, we can also solve this problem in $O(K)$ time per test too, without using $O(K^2)$ pre-computation, enabling solutions for $K \leq 10^5$ too, but that is unnecessary here.

As a closing comment, I have to say, many people might have just used oeis, and if that site didn't exist, the number of ac submissions might have been lower.

Time Complexity : $O(K^2 + T \cdot K^2)$

Also, you can read about multiple different solutions to this problem in the comments section below, check it out.

PREREQUISITES:

Backtracking, Combinatorics, and Modular Arithmetic.

PROBLEM:

Given an array A of length N with some elements missing and a value S, consider all ways of filling missing elements and for every resulting array, find the sum of gcd of every unordered pair. Print the sum modulo $10^9 + 7$.

SUPER QUICK EXPLANATION

- Order of elements does not matter. If C is the number of missing elements in the array and Si is the sum of non-missing elements, We need to fill C values so that their sum is $S - Si$.
- We can calculate all such ways by forming all ways, say in the array B of length C, such that $B[i] \leq B[i+1]$ and counting all different ordering using combinatorics.

EXPLANATION

Considering the slow solution first.

We can make a backtracking solution where we try to fix the first x elements, and filling missing elements with all possible choices such that the total sum does not exceed S. Then iterating over every pair of values and calculate pairwise gcd-sum and print the answer. Time complexity, in this case, is Exponential, thus, will pass only the first subtask.

For the second subtask, we need to work harder than that.

First of all, Notice that the order of elements does not matter at all since the position of an element is irrelevant to our problem.

Second thing, If Si is sum of visible elements, Missing elements can have sum $S - Si$. Now, Suppose we have a parititon $B_1, B_2, B_3 \ldots B_k$ such that $\sum B[i] = S - Si$.

If we can get all partitions of $S - Si$ into C elements, we can simply calculate the answer in $O(N^2)$ by iterating over every pair and taking gcd.

But, Number of partitions of S is also very high if we consider them ordered. Like, If we consider $1 + 2$ different than $2 + 1$ as a partition of 3, Number of the partition of 50 is very high.

But, We can notice, that pairwise gcd-sum do not change, when just the ordering of elements of partition changes. Like, Suppose we have example N = 3, S = 4 and array 1 -1 -1. We can notice that pairwise gcd-sum of 1 1 2 is same as pairwise gcd-sum of 1 2 1. Hence, if we can calculate the pairwise gcd-sum for all distinct partitions of $S - Si$ with non-missing elements, we can solve this problem fast enough.

Now, we can once again use backtracking solution, this time with another constraint, that current element cannot be smaller than the previous element. This way, we can get a partition and we calculate pairwise gcd-sum.

But, the problem is, how do we know how many times a partition will appear. Fortunately, we can **rush to Combinatorics** 😛 for answers. We can see, It is just the number of Distinct permutations of the given sequence.

Fortunately, If any element occurs x times, the Same permutation will be counted x times. This way, We can just Find the number of distinct permutations of the given sequence, and thus, find the required answer.

Hence, we can just find every unordered partition of $S - Si$ into C elements, Find pairwise gcd-sum with each partition and also the number of times the same partition may appear and calculate the answer.

For reference, the method of computing distinct permutations of a sequence can be found here [6] .

Time Complexity

The time complexity of this solution is $O(P(S) * N^2)$ where $P(S)$ is the number of unordered partitions of S. There is no closed form for Number of unordered partitions of a number, but for $S = 50$, it is small enough for our solution to pass comfortably.

PREREQUISITES:

Precomputation, Bitmask, Dynamic Programming, Combinatorics.

PROBLEM:

Given a tree of N nodes, Each node can have either 0 or 1 coin. Find number of ways to select nodes to put coins such that GCD(X, Y), where X, Y represent number of coins in subtree of any two disjoint subtrees.

QUICK EXPLANATION (maybe quick but cannot be complete)

- BitMasking on prime numbers up to $N/2$ is done for building DP states.
- Dynamic Programming state is represented as (u, X, M), representing Number of po
- DP state is calculated from leaf to root, Every combination of mask is checked and numb
- Final answer is represented as sum of all possible combinations of Number of coins and

EXPLANATION

This is going to be a long editorial, so, be prepared. 🙂 Here we go!!

First of all, to handle **GCD constraint**, we need to keep information of all the primes, which have been used in some subtree. We can keep this information in terms of primes which divide any of the subtree.

There are 19 primes between 1 to 70, but we ask ourselves, do we need all this information?? Where would we use the information that a subtree is divisible by 37, **when there cannot exist another disjoint subtree having size divisible by 37, as it would require atleast 75 nodes in tree (not 74)**.

So, we need only primes upto $N/2$, and there are 11 primes. Let's create a bitmask over all these primes.

If ith bit of mask is on, it implies that current subtree or subtree of any descendent of current node has number of coins divisible by ith prime.

Now, If i say a mask of subtree of node, then it should be understood that it means that subtree of node, or its descentants' subtrees are divisible only by primes in this mask. This definition would be used in editorial.

Now, move towards Dynamic Programming.

I'll Discuss the slow approach first and then woudld see how to optimize it to fit within Time Limit. Time complexity analysis has been given at the end.

Let us define two 3D lookup tables, $ANS[i][j][mask]$ and $DP[i][j][mask]$, defined as follows.

$ANS[X][j][mask]$ denote number of ways to put excatly j coins in subtree of X including X, such that the mask of subtree of is $mask$. This table stores final answer for every combination of (Node, coinCount, mask).

If node X is leaf, we can either put the coin in subtree or keep it empty. But ways are represented as $ANS[X][0][0]$ and $ANS[X][7][0]$ respectively. Assign these two to 1 and rest will remain 0. (0 or 1 cannot be divisible by any prime).

If node X is a non-leaf node, we use Dynamic programming to find out number of ways to put j coins in subtree of X excluding X. The DP state can be represented as

$DP[i][j][mask]$ is defined for a node X, the Number of ways to put j coins in subtree of First i children of X such that the mask of nodes is $mask$.

$DP[0][0][0] = 1$ Because we can put 0 coins in first 0 children of node X such that their mask is 0. i.e. $BitWise_AND(mask1, mask2) == 0$

Transition in DP table would be represented by moving from ith child to (i+1)th child as follows.

$DP[i+1][coins+add][mask1|mask2] += DP[i][coins][mask1] * ANS[child][add][mask2]$ if mask1 and mask2 do not share any prime.

For all combination of (i, coins, mask1, add, mask2) for each child of X. i is just Child number in 1-based indexing and coins is sum of subtree sizes of previous i children.

Using this DP, we will get final states in $DP[outDegree(X)][j][mask]$ representing Number of ways to put j coins in subtree of X exculding X such that their mask is $mask$.

After above DP, we need to update ANS table with answer for node X for every mask, handling cases whether Coin is put at node X or not separately by iterating over each combination of (number of coins, mask) covered in DP[outDegreee(X)] and updating answer and mask carefully.

This solution will get TLE result. Time complexity of this solution has been discussed at the end.

Optimizing above solution

First of all, if, For any combination of (i, coins, mask1) if $DP[i][coins][mask1]$ is zero, we can directly skip these without iterating over (add, mask2)

Second thing, In case we have a combination where add is divisible by any prime with bit 1 in mask1, it would definitely voilate the GCD constraint, so we have to skip it.

The most important thing is, the $BitWise_AND(mask1, mask2) == 0$ constraint. At present we are iterating over $2^{11} * 2^{11}$ combinations, which dominates the solution complexity and is the main cause of TLE.

Can we think about any way to directly iterate over only such combinations satisfying above constraint so that to reduce above time.

Thankfully, there exist a method [8] by means of which we can directly iterate over submasks of a mask. We can utilize the fact that all values of mask2 satisfying $BitWise_AND(mask1, mask2) == 0$ are submasks of XOR(2^{11}, mask1). This results in $O(N^2 * 3^{11})$ time complexity which shall easily pass within time limit.

Click to view

Let mask1 and mask2 be the two given masks.Above technique is applied as

```
supermask = ((2^11)-1)^mask1

for(int mask2 = supermask; true; mask2 = (mask2-1)&supermask){
    //Do your work with current mask2
```

```
    if(mask2==0)break;
}
```

Though following trick isn't necessary, if can come in handy if memory is a constraint and occurs quite frequently in Dynamic Programming problems.

In above case, for DP table ith transition ith, we need information of only (i-1)th layer, so, we can reduce memory consumption of DP table by a factor of N.

Time Complexity Analysis:

For First Solution, For every node, we iterate over every combination of masks, taking $2^{11} * 2^{11} = 4^{11}$. We repeat this procedure for OutDegree(i) times pref[i-1, seemingly an $O(N^2)$ structure, implying around $O(n^2 * 4^{11})$ iterations per node, but the summations of Out degree of all nodes is bounded by N, so overall time complexity is amortized to $O(N^2 * 4^{11})$

For optimized solution, For Every mask having j bits 1, we iterate only over the masks which have those j bits off. Specifically, for every mask with j bits on, we iterate only over $2^{(11-j)}$ masks in worst case. We have exactly $C(11, j)$ such masks.

So, total complexity of iterating over masks become $\sum_{i=0}^{11} C(11, j) * 2^j$ which is just the binomial expansion of $(1 + 2)^{11} = 3^{11}$.

The Final Time complexity becomes $O(N^2 * 3^{11})$.

I know that the editorial is long and problem is compliacated, so feel free to ask any doubts, queries or point out any mistakes.

Prerequisites

Factorisation, Combinatorics

Problem

You are given N workers and each of them has a deadline to complete the work, given by d_i for i^{th} worker. Each worker completes the work in 1 day and on each day at most 1 worker can work. You need to decide to the array containing the deadline for every worker such that the number of ways in which the workers can decide to work is C and d_n is minimised. Also, the array of deadlines should be sorted.

Explanation

Before proceeding towards the solution, we need to find the number of ways in which workers can decide to work for a given array $d_1 \leq d_2 \leq \cdots \leq d_n$.

Consider the case for $n \leq 3$. Let $d_1 = x, d_2 = y, d_3 = z$. We have the following dynammic programming solution

$$dp[x] = x$$

$$dp[x][y] = dp[x] * (y - x) + dp[x - 1] * x = xy - x^2 + x^2 - x$$

$$dp[x][y] = xy - x = (y - 1) * dp[x]$$

$$dp[x][y][z] = dp[x][y] * (z - y) + dp[x][y - 1] * (y - x) + dp[x - 1][y - 1] * x$$

$$dp[x][y][z] = (xy - x)(z - y) + (xy - 2x)(y - x) + (xy - 2x - y + 2)x$$

$$dp[x][y][z] = (xyz - xz - 2xy + 2x) = (z - 2) * (xy - x)$$

$$dp[x][y][z] = (z - 2) * dp[x][y]$$

Using the above observations, it is clear that number of ways is $\prod_{i=1}^{i=n} (d_i - i + 1)$. We can even prove it combinatorially. The first person has d_1 choices, the second one has $(d_2 - 1)$ choices and so on. By the principle of multiplication, the number of ways comes out to be same.

Note the above solution clearly points out that $d_i >= i$. Thus for $C = 1$, the array is $(1, 2, 3, 4, \cdots)$.

Thus, the problem now reduces to finding an array $d_1 \leq d_2 \leq \cdots \leq d_n$ such that $\prod_{i=1}^{i=n} (d_i - i + 1) = C$ and d_n is minimised.

Instead of finding $d_1, d_2 \cdots d_n$, we will find sorted array, $d_1, (d_2 - 1), \cdots, (d_n - n + 1)$, such that product of this sequence is C and later add back $(i - 1)$ to every term to ensure that sequence is increasing.

Subtask 1: N ≤ 10, C ≤ 100

We can use dynamic programming based solution for this subtask. Let $dp[N][C]$ denote the minimised value of d_n for which the sequence $d_1, d_2, \cdots, d_n$ has number of ways as C. The transitions are simple. Let $dp[N][C] = x$. So, we need to find the minimum value for the sequence with $(N-1)$ terms and product as (C/x). The possible candidates are the divisors of (C/x). So, we try all possible candidates and update our answer whether or not it is possible to form an array with the tried candidate. Once, the dynamic programming table is built, we can simply backtrack it to print the answer.

The time compleixty of the above approach is $O(N * \sigma_0(C)^2)$, where $\sigma_0(C)$ denotes the number of divisors of C.

For details, you can refer to author's solution for the above approach.

Subtask 2, 3: N ≤ 10^6, C ≤ 10^9

We may see that for optimal answer, the sequence $d_1, (d_2 - 1), \cdots, (d_n - n + 1)$, looks like the following for large n :

[zero or more numbers greater than 1] [bunch of ones] [zeros or more numbers greater than 1]

Outline of proof for above claim: Basically we want to show that if we have a solution that is not in the above form, then there's a solution of the above form which has the same last element. We achieve this by moving the blocks of non 1 elements as a unit, in this part, we will use the monotonicity of array d as an argument to show that the blocks are indeed movable and don't break anything.

The above form is important because increasing N actually just pushes more ones into the middle (if there were a better way to distribute the numbers we would have used it earlier). So to solve, we calculate the optimal sequence for smaller N and then extend the "bunch of ones" in the middle.

In the worst case the above method's complexity is $O(N + (\text{number of prime factors of C}) * \sigma_0(C)^2)$. Because the number of prime factors in the range of 10^9 is 30 in the worst case and the number of divisors is 1000 in the worst case, this gives around $3 * 10^7$ operations per test case.

For more details, you can refer to the setter's or tester's solution below.

Editorialist Solution: Based on solutions by contestants in the contest (Greedy approach)

Incorrect Solution: Note that the below solution is wrong for small values of N. This is because it is not guaranteed that choosing the largest factor at every moment in the below solution will yield an optimal solution. It though will work for cases where $N > \log C$ as even the greedy approach will take $O \log C$ steps to check optimality for the answer. For details refer to the below comments for some counter cases. Note that in all the test case $N \leq \log C$.

We will first store all the factors of C. Now, we traverse the factors one by one and check if we can form an array $d_1, (d_2 - 1), \cdots, (d_n - n + 1)$, such that the last element is given factor. Is yes, we simply update the answer. Below is the pseudo-code for checking whether we can find an array with the last element as given factor x.

```
# facts store the factors of C.
# facts[idx] = x

def check(idx):
    ptr = n
    prod = c
    while ptr > 0:
        while prod % facts[idx] != 0
```

```
            idx -= 1
        prod /= facts[idx]
        ans[ptr] = facts[idx] + ptr - 1
        ptr -= 1
        if idx != len(facts)-1 and facts[idx] == (facts[idx+1] - 1):
            idx += 1
        if idx == 0 or prod == 1:
            break
    return (prod == 1)
```

Let us analyse the above pseudo-code. We first try to greedily find the first largest factor of the remaining product. At first step it will be $facts[idx] = x$, the number we want to check. Since in final array we want $d_{(i-1)} \leq d_i$ and we are finding array $(d_i - i + 1)$, the previous term can now contain a possible larger factor only if it is greater than the previous factor by 1. The last condition just checks that if the product has become 1, then we know the trivial answer or if the factor to be considered is 1, the product will remain same. Below is an example iteration of the above for $N = 8$ and $C = 36$. Let the factor to be checked is $x = 2$.

$$\text{facts} = 1, 2, 3, 4, 6, 9, 12, 18, 36$$

$$\text{Step } 1 = \cdots, 2$$

$$\text{Step } 2 = \cdots, 3, 2$$

$$\text{Step } 3 = \cdots, 3, 3, 2$$

$$\text{Step } 4 = \cdots, 2, 3, 3, 2$$

Since the prod now becomes 1 we break. To build the required array we add the offsets $(i - 1)$ to the array.

Thus, required array is $(1, 2, 3, 4, 6, 8, 9, 9)$. Note that this may or may not be the optimal solution. It just shows how we check if a given factor can form the series if it is the last term in the sequence.

The time complexity of the above approach will be $O(\sigma_0(C)^2 + N)$. This is because, for every factor, the checking will take $O(\log C)$ step in the worst case as “prod” variable will decrease by a factor of atleast 2 in each iteration. But the dominating factor will be the while loop which determines the optimal factor in each iteration and can take up to $\sigma_0(C)$ operations. The second term, $O(N)$, is for building the complete sequence containing the initial trivial part of $(1, 2, 3, \cdots)$ and printing the answer. The space complexity will be $O(N + sqrt(C))$.

Once, you are clear with the above idea, you can see the editorialist implementation below for help.

Feel free to share your approach, if it was somewhat different.

Time Complexity

$O(N + (\text{number of prime factors of C}) * \sigma_0(C)^2)$

Prerequisites: Combinatorics

Problem: You have a bunch of cycles, each containing some nodes. For each node, starting from the node, travel across the cycle until you reach the same node. Now, you have an array `A` of N elements, which means that the `i`th node occurs `A[i]` times after applying each operation. You need to tell how many ways are there to form those cycles that are consistent with array `A`.

Explanation: Let's clear up the problem statement given on the contest page. Gritukan writes a permutation `P` of `N` numbers such that, for each node i, visit `P[i]`, then `P[P[i]]`, and repeat this until you reach `i` again. Now, this guarantees that the permutation has each node in a cycle exactly once. Proof?

1. Each node is a part of a **cycle**, otherwise we will never reach the start node again.
2. Each node is part of a cycle **exactly once**, since we can move to only one node from a current node.(Just think about it!)

We need to find how many such permutations occur, given how many times a node has been visited in Gritukan's scheme.

Graphically looking, any permutation would result in a bunch of different sized isolated cycles. Talking about just one cycle having `p` nodes, obviously each node of that cycle will occur `p` times in Gritukan's scheme (once for each time the algorithm starts for every node). Hence, if `A[i]=p`, that means node `i` is a part of a cycle of size `p`. Now, keeping that in mind, let us prove one fact that for a valid permutation to occur, in the array A, `p` will occur `k*p` times where `k` is any arbitrary positive integer(otherwise, there will always be some number of nodes less than `p` left that cannot be satisfied). Hence, for a valid permutation to occur, the only condition is each number `p` present in the array `A` must be `k*p` times in the array.

Now, all the concept behind this problem is over. It all comes down to the math: You have to form k_i cycles of p_i nodes each, such that the total number of times p_i occurs, $m_i=k_i.p_i$. This can be given by the formula:
$\prod \frac{\left({}^{m_i}C_{p_i}.(p_i-1)\right).\left({}^{m_i-p_i}C_{p_i}.(p_i-1)\right).\left({}^{m_i-2p_i}C_{p_i}.(p_i-1)\right)...\left({}^{p_i}C_{p_i}.(p_i-1)\right)}{k_i!}$ where ${}^nC_r$ means $\frac{n!}{r!.(n-r)!}$.

We select `p` nodes from `m` nodes in ${}^mC_p$ ways, arrange them in a cycle by `p-1` ways, then repeat it again, until all nodes are put in a cycle. At last we divide by k!, because each cycle is identical.

For my solution, I have broken down the formula to this:
$\prod \frac{1}{k_i!}.\frac{1}{p_i^{k_i}}.(\frac{p_i!}{0!}.\frac{2p_i!}{i!}.\frac{3p_i!}{2i!}...\frac{m_i!}{m_i-p_i!})$

Complexity and Implementation: Computation of how many times each number comes in array `A` can be done in linear time. Then, we iterate over each p_i and get m_i. Then, loop through m_i to p_i at a decrement of p_i, and calculate the answer. Although nested loop, the complexity of this actually of the order O(N). Since, at max, we will have to iterate from m_i to 1, at decrement 1. However, the sum of m_i over `A` is actually `N`.

We will be actually computing the value of all the factorials%MOD beforehand, along with their inverses, for constant time access. This build operation will take at most 10^6 operations.

editorialist's solution 29

PREREQUISITES:

Dynamic Programming 1 , Combinatorics 4

PROBLEM:

The problem can be reduced to the following grid world scenario.
Chef is at coordinate $(0, 0)$ in a 2-D grid and has to reach point (p, q). From any point (x, y) he can move only in increasing direction: either UP(to the point $(x, y + 1)$) or RIGHT(to the point $(x + 1, y)$). Here, UP & RIGHT correspond to a bad & good deed respectively. But he can move UP if he is strictly below the line $x = y + c$. M points are blocked in the grid(he can't visit a blocked point), whose coordinates are given.
You have to count number of such paths possible. Two ways are same if and only if path taken is exactly same in both ways. Print the required answer modulo $10^9 + 7$.

QUICK EXPLANATION:

The problem can be broken down in 2 parts.

1. Make a function that finds the number of ways to reach point $(x2, y2)$ from point $(x1, y1)$ with subject to the constraint that we can't cross line $x = y + c$.
2. The number of ways of reaching any blocked point (i,j) is independent of any blocked cell having larger x or y coordinates. If we sort the blocked cells on basis of x,y in increasing order - the number of ways to reach a cell in ith index of the array is independent of any cell after it in the array. So we can find the number of ways reaching any blocked cell by using m^2 DP approach.

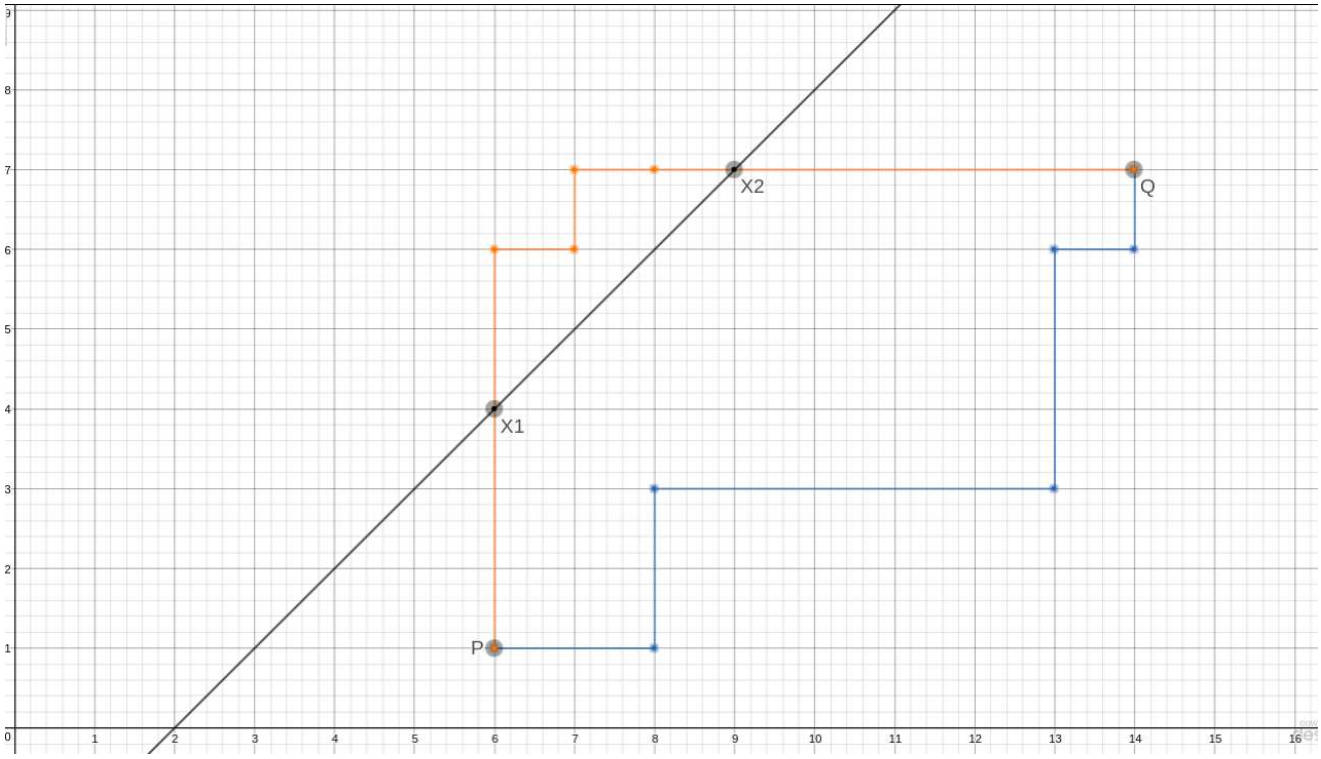
The overall complexity is $O(M^2 + p + q)$.

GENERAL SUGGESTION:

This editorial will have some hand exercises. It's highly suggested that you yourself try to figure out the solution to those before proceeding further.

EXPLANATION:

Part 1: *Find number of ways to reach point $Q(x2, y2)$ from point $P(x1, y1)$ if $x2 >= x1$ and $y2 >= y1$ without crossing the line $x = y + c$ and ignoring the blocked cells:*
Consider the following figure.



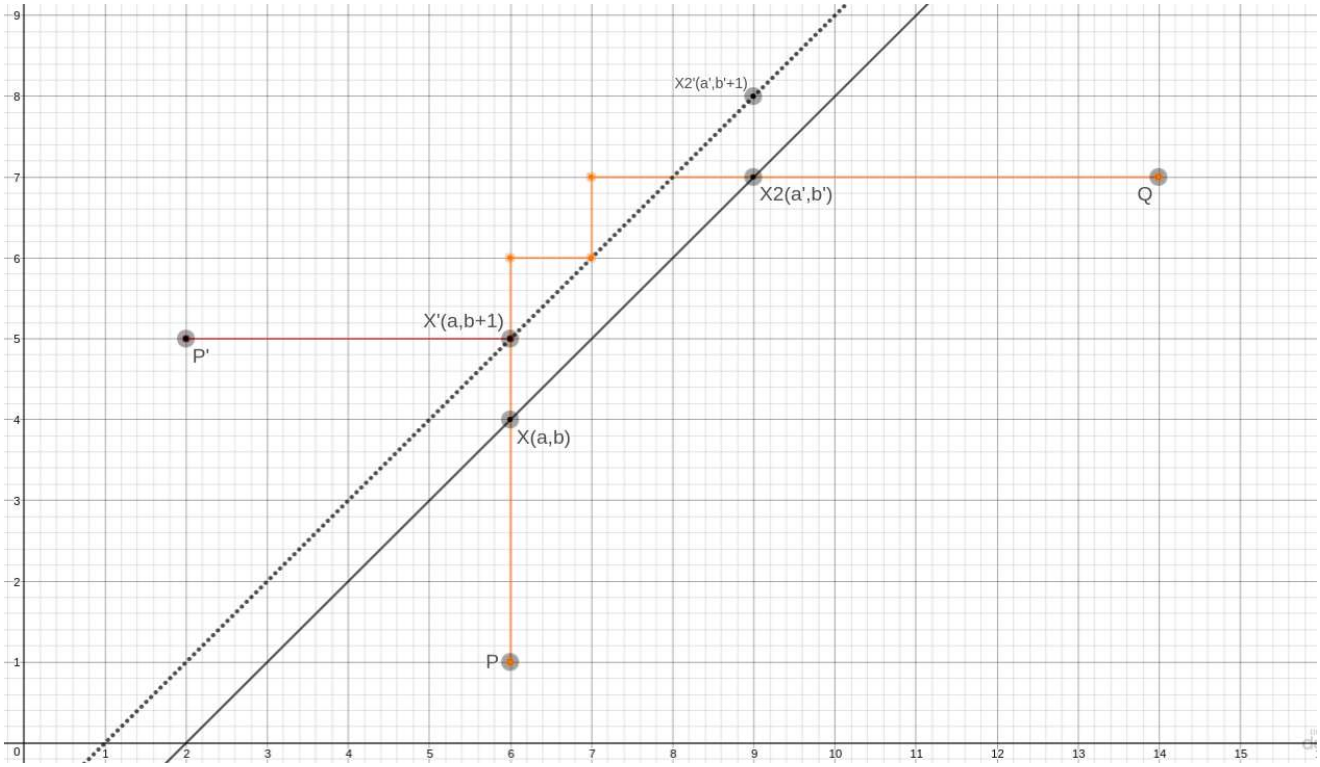
Here we want to go from P to Q without crossing the line(black line). We will call it L1.
Consider all possible paths from P to Q. There can be 2 types of paths:

1. Paths from P to Q without crossing the line L1. We call such a path a *good path*. **NOTE** A path is good as long as it doesn't cross L1. Although, it can it touch it any number of times.
2. Paths that cross the line L1. We call such a path a *bad path*.

In the figure the blue path is a good path whereas the orange path is a bad path.
Now the sum of all good paths and bad paths gives us the total number of paths from P to Q.

`Ex1: Find total number of paths from point P(x1, y1) to point Q(x2, y2) where x2 >= x1 and y2 >= y1.`

Let $x2 - x1 = x$ and $y2 - y1 = y$. The total steps needed to be taken is $x + y$. Also, it's fixed that each path will have exactly x RIGHT steps and exactly y UP steps. Different paths can be generated by permuting the UP and RIGHT steps amongst them. The total number of ways to permute $x + y$ items such that x and y of them are identical respectively is given by: $(x + y)!/(x!y!)$ where $n!$ denotes n factorial.
Now, we have found the total number of paths from P to Q. But our aim was to find the total number of good paths from P to Q. We can first find the total number of bad paths and then subtract is from the total number of paths. This will give us the total number of good paths from P to Q.
For a path B to be bad, it has to cross L1 at atleast 1 point. Now, if it touches the line L1 at point (a, b) after crossing L1, it will reach point $(a, b + 1)$.



Any general point on L1 is $x = y + c$, so any general point that a bad path reaches on crossing L1 is given by $x = y + c - 1$. Let this line be L2(black dotted line). So we can say that any path B that touches L2 atleast once is a bad path.
Let the first point where B touches L2 is X'. Now take reflection of the segment of B from P till X' about the line L2 as shown in the figure in green color. Consider a new path B' : formed by the reflected segment of B and the rest of the unreflected segment of B from X' to Q i.e. the path from P' to X' + the path from X' to Q.

Now there are 2 important observations about the path B'

1. Starting point of B' is always point P' i.e. reflection of P in L2.
2. B' is a path from P' to Q that always touches the line L2 atleast at one point.

Now, we easily prove by construction that for every bad path B, that touches the line L2 for the first time at X', we can construct a corresponding path B' from P' to Q by : concatenating the reflected segment of B from P to X' and the segment of B from X' to Q.
Also, by similar argument, for every path B' from P' to Q we can construct a bad path B by again concatenating the reflected segment of B' from P' to X' and the segment of B' from X' to Q. So there is a bijection between bad paths B and the paths B from P' to Q.
Now, to find the number of bad paths we just have to calculate the total number of paths from P' to Q. So all we are left to do is to find the reflection of P in L2.

```
Ex2: Find the reflection of P in L2.
```

the image of the point (x_1, y_1) in the line $ax_1 + by_1 + c = 0$ is given by:

$$\frac{x - x_1}{a} = \frac{y - y_1}{b} = \frac{-2(ax_1 + by_1 + c)}{a^2 + b^2}$$

The above formula can be looked through in any high school book. Using it, reflection of $P(x1, y1)$ in L2: x = y + c - 1 is $(y1 + (c - 1), x1 - (c - 1))$. Now number of paths from $(y1 + (c - 1), x1 - (c - 1))$ to $Q(x2, y2)$ can be found using the formula above.
Now, by subtracting it from the total number of paths from P to Q we can get the total number of good paths from P to Q as was required. Let $F(x1, y1, x2, y2)$ gives us this number.

So the final expression for number of paths from P(x1,y1) to Q(x2,y2) without crossing L1: x = y + c is given by
$Ways = (x1 - x2 + y1 - y2)C(x1 - x2) - (x1 - x2 + y1 - y2)C(x1 - x2 + (c - 1))$.

Now, we have the solution to the first part. But wait! there is one more small exercise for you.

```
Ex3: How to calculate C(N,K) for N, K <= 4 * 10^6?
```

Sol: We can pre-compute and store all the factorials and inverse-factorials for $1 <= i <= 4e6$. Now to compute C(N,K) all we have to do is 2 elementary operations:

```
long long findnCk(N, K) {
    ll nCk = fact[N];
    nCk = (nCk * inv[K]) % MOD;          //where inv[K] is modular inverse of K
    nCk = (nCk * inv[N-K]) % MOD;
    return nCk;
}
```

But how to compute factorials and inverse-factorials of very large numbers?
Finding factorial for very N can be done in O(N) by simply using the factorial of previous index

```
fact[i] = (i * fact[i-1]) % MOD;      //where fact[0] = 1
```

Now, we have to efficiently find the inverse-factorial. We can this too, using the inverse-factorial of next index as follows.

```
inv[i] = (inv[i+1]*(i+1))%MOD;   //where inv[4e6] is modular-inverse of fact[4e6]
```

So the complexity of this pre-computation is O(N) or O(p + q). These are some relevant links: Modular multiplicative inverse
,Fast Exponentiation ,A small intution of the approach. [1]
Now let's, put $P(x1, y1) = (0, 0)$, $Q(x2, y2) = (n, n)$ and making $L1 : x = y$ i.e. $c = 0$ in the above formula, we get $2nCn - 2nC(n - 1)$ which is the expression for nth **catalan number**(sequence of natural numbers occurring in various counting problems).

Awesome! This just shows how the catalan numbers can be derived. Now, you are good to go with all catalan related problems. 🙂

```
Ex4: What is the number of full binary trees with n internal vertices?
```

Part 2: *Find number of ways to reach point $Q(x2, y2)$ from point $P(x1, y1)$ without going to blocked cells.*

The naive way to incorporate the blocked cells is using **Inclusion-Exclusion Principle**. Let (x, y) be the starting point. The number of ways to get from (x, y) to (m, n) while avoiding the one restricted point at (a, b) is given by the number of ways to get to (m, n) with no restrictions, minus the number of ways to get to (m, n) that go through (a, b).
$F(x, y, m, n) - F(x, y, a, b) * F(a, b, m, n)$
Generalising, it we can get the total number of ways to reach (m,n) without going to the blocked point.

```
Ex5: How? This exercise is left for the user.
```

But, such a solution will take 2^m time where $m = 10^3$. Hence, not feasible. We need to improve.

It is subtle observation that the number of ways of reaching any blocked point (i, j) is independent of any blocked point having larger x or y coordinates. So we can sort the blocked cells on basis of the pair (x, y) in increasing order so that the number of ways to reach a cell in ith index of the array is independent of any cell after it in the array. So we have broken the given bigger problem into **smaller subproblems**.
Nice enough? Uhhh! wait we also have an **optimal substructure** in the problem. Let the point P_i at ith index is (x_i, y_i).
Actual number of ways to reach

$P_i = (waysToReachP_iIgnoringTheBlockedCells) - (\Sigma(waysToReachFromP_iToP_j) * (waysToReachP_jIgnoringTheBlockedCells))$.

Hurrah! Now we have a straightforward M^2 dynamic programming solution.

Let's say our set $S = allBlockedCells + cell(p, q)$. Sort S on increasing basis of x coordinate and then increasing on y. Also let's assume $dp[i] = F(x, y, x_i, y_i)$ where (x, y) is the starting point of the path. Pseudo code for it is given below:

```
for(i = 0; i < S.size(); ++i)
    for(j = i-1; j >= 0; --j)
        if(S[i].x >= S[j].x && S[i].y >= S[j].y)
             dp[i] -= (dp[j] * F(S[j].x, S[j].y, S[i].x, S[i].y));
```

The overall complexity is $O(M^2 + p + q)$

AUTHOR'S AND TESTER'S SOLUTIONS:

Author's /Editorialist's 17

Tester 3

RELATED PROBLEMS:

Part 1

UVA - 932 Safe Salutations

Hackerrank - devu and cool graphs 1

Codechef - Hand Shake 3

Codechef - Mock Turtle 1

Part 2

Codeforces - Count Ways 7

Codeforces - Research Rover 4

PREREQUISITES:

DP, Permutations and Combination basics

PROBLEM:

The questions asks you to find the maximum value of N such that the total number of ways of selecting soup bowls would be less than a given number max.

QUICK EXPLANATION:

You simple need to form the combination used to find the number of ways for each N. You can see that the N will not exceed value 25 for any max up-to 10^9. Hence you can easily iterate over N until you get its maximum value as per the condition given.

EXPLANATION:

First of all, we have to construct a table with the combination values. Here, we have to simply consider values up-to 25 so that we can get the values of each combination in $O(1)$. Even it is possible to calculate the value of each combination individually every time because the N is not so large. The combination table can be generated as per following snippet:

```
void getCombinations() {
    ll i, j;
    C[0][0] = 1;
    for(i = 1; i < N; i++) {
        C[i][0] = 1;
        for(j = 1; j <= i; j++) {
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
}
```

On constructing we can get the combination in $O(1)$ time. Further looking at the problem, we know that we have to select at least 2 soup bowls with different variety. Also, we don't necessarily need to select N bowls. Hence we can select the bowls which are at least 2 and up-to N. It is required that 2 bowls must be of different variety. Hence, we can select those 2 from N by ${}^nC_2$. Now, say we decided to select 3 bowls from N out of which we know that 2 must be of different variety and the one can be any of the N varieties. This can be done in ${}^NC_2 +^N C_2 *^N C_1$. Hence, we can simply reduce this formula for selecting M such soups like ${}^NC_2 +^N C_2 *^N C_1 +^N C_2 *^N C_2 + ... +^N C_2 *^N C_M$. We can get each of the combination values for the table created. While iterating over to values of M from 1 if we find that the number of ways at certain M is greater than the max we can stop iterating and print the value of $M - 1$ there.

ALTERNATIVE SOLUTION:

Could contain more or less short descriptions of possible other approaches.

Prerequisites:

Combinatorial Game, Greedy

Problem:

Dee and Dum play modified nim game on **N** stacks of stones. Each stone can be of two types - 0 or 1. Dee can remove stones from a stack whose topmost element is 0. And Dum can only remove from stack with 1 at the top. If both play optimally who will win?

Explanation:

Note that to win the game, every player will try to pick stones such that number of moves for him is as large as possible before the end of the game. And, number of moves for the other player is as little as possible. This way he can make the other player run out of moves and can win.

How to achieve that?

Consider Dee. He can pick stones from stacks with top element as 0. So in every step, he will have to remove at least one 0-stones. How about picking exactly one 0-stone in each step? This way we see that he will waste no stones which are useful to him. Also, he can try to waste some of the stones of the other player by removing all consecutive 1-stones just below the top 0-stone. This gives him multiple advantages.

First, he picks no more than one 0-stone. Second, after removal of those stones, the top stone in the stack is still 0-stone. This means in future moves he can use this stack. Third, he wastes some 1-stones which are useful for the other person.

Using this strategy, he can make sure that all stacks whose top element is 0-stone will be used only by Dee, similarly other stacks will be used by Dum.

How many total moves Dee can make? It is number of 0-stones in all the stacks with top stone as 0-stone. Call this number **stoneCountDee** . Similary, let's define **stoneCountDum** as number of 1-stones in the stacks with top element as 1-stone.

If Dee moves first, he can win iff **stoneCountDee** $>$ **stoneCountDum**.

If Dum moves first, he can win iff **stoneCountDum** $>$ **stoneCountDee**.

PREREQUISITES:

Math, Combinatorics, Pascal Triangles

EXPLANATION:

The sequence in the question is the generalisation of Moessner sequencce. In Moessner sequence $d_1 = d_2 = ...d_{m-1} = d_m$.
Consider $N = 8, m = 2, d_1 = d_2 = 4$

1 2 3 4 5 6 7 8
1 3 6 11 17 24
1 4 15 32
1 16

As we can observe every group satisfy the Pascal property (each interior element is the sum of the elements immediately above it and to its left).
Lets add a new row and column of 1's in the first group.

1 1 1 1 1
1 2 3 4
1 3 6
1 4
1

Pascal Traingle

Similarly for Second group add a new row of 1's and a column containing previously marked numbers,

1 1 1 1 1 1
4 5 6 7 8
6 11 17 24
4 15 32
1 16

Now we will represent the above sequence algebraically using Pascal triangle.
Let $h_k(x, y)$ be the algebraic representation of the kth triangle.
So, $h_k(x, 1)$ will be the polynomial having coefficients as the marked numbers for the kth triangle then,

Each triangle is obtained from the previous by taking the homogeneous component of degree $diff_k$, and multiplying by $\Delta(x, y)$ and then putting $y = 1$ in the obtained triangle.

$h_{k+1}(x, y) = [h_k(x, 1) * \Delta(x, y)]_{d_k}$ and $h_0(x, 1) = 1$ where d_k is the size of group k and

$\Delta(x, y) = \frac{1}{1-(x+y)} = (x + y)^0 + (x + y)^1 + (x + y)^2 + ... = \sum_{d=0}^{\infty}(x + y)^d$

(Note:-$\Delta(x, y)$ is the algebraic representation of pascal triangle.
The nth north-east to south-west row of the pascal triangle will be:-
$[[\Delta(x, y)]_{n-1}]_{y=1} = (1 + x)^{n-1}$)

which simplifies to

$h_k(x, 1) = \prod_{i=0}^{k-1}((k - i) * x + 1)^{diff_i}$ where $diff_i = d_i - d_{i-1}$ and $d_0 = 0$

Now what we require is the sum of all marked numbers in the original group which is:$h_k(x, 1) - 1$ (subtracting 1 as we added the rows of 1's ,so 1 needed to be subtracted)

Final ans:-
$\sum_{k=1}^{m}(h_k(x,1)-1)$

Eg:-
Consider $n=4, m=2, d_1=d_2=2$
now $h_0(x,1)=1$
now
$h_1(x,y)=[h_0(x,1)*\Delta(x,y)]_2=[1*((x+y)^0+(x+y)^1+(x+y)^2+...)]_2=(x+y)^2$
$h_2(x,y)=[h_1(x,1)*\Delta(x,y)]_2=[(x^2+2x+1)(1+x+y+x^2+y^2+2xy+...)]_2=4x^2+4xy+y^2$
now $h_1(x,1)=1+2x+x^2$ represents the first modified triangle

```
1 1 1
1 2
1
```

and
$h_2(x,1)=1+4x+4x^2$ represents the second modified triangle

```
1 1 1 1
2 3 4
1 4
```

TIME COMPLEXITY:

$O(m*m*log(max(diff[i])))$

PREREQUISITES:

Math, Combinatorics, Generator Functions

PROBLEM:

Maximise A_0 XOR A_1 XOR A_2 ... XOR A_{N-1}, where $A_i \in (1, 2, 4, ..., 2^{k-1})$, and find no of different ways to do it modulo $10^9 + 7$

EXPLANATION:

We'll split the solution into two parts:-

Part 1: $N-K$ is even

All the numbers in S must occur odd number of times Number of ordered solutions of this is $\sum \frac{N!}{x_1!x_2!...x_k!}$ over all $x_1+x_2+...+x_k=N$ where x_i is odd. This is equivalent to the coefficient of x^N in $\left (\frac{x}{1!} + \frac{x^3}{3!} + \frac{x^5}{5!} + ...\right)^k$ The generator function of the above polynomial is $\left (\frac{e^x - e^{-x}}{2} \right)^k$ Thus the solution is the coefficient of x^N of the above eqn, which on expanding binomially we get $\sum_{r=0}^{k} \frac{\left (-1 \right)^r \binom{k}{r}e^{(k-2r)x}}{2^k}$ Coefficient of x^N in $e^bx = \frac{b^N}{N!}$, thus coefficient of x^N in the above eqn will be, multiplied by $N!$, is $\sum_{r=0}^{k} \frac{\left (-1 \right)^r \binom{k}{r}(k-2r)^{n}}{2^k}$ which is the final answer to Part 1.

Part 2: $N-K$ is odd

Coefficient of 1 is to be kept even, while rest of the coefficients are odd Polynomial for even x is $\frac{x^0}{0!}+\frac{x^2}{2!}+\frac{x^4}{4!}+...$ Whose generator is $\left (\frac{e^x + e^{-x}}{2} \right)$
Thus the final expression, with 1 even generator and $k-1$ odd generators are

$$\left(\frac{e^x+e^{-x}}{2}\right)\left(\frac{e^x-e^{-x}}{2}\right)^{k-1}$$

Which leads us to the final solution is, after reducing similarly as above:-

$$\sum_{r=0}^{k-1} \frac{(-1)^r \binom{k-1}{r}((k-2r)^n+(k-2r-2)^n)}{2^k}$$

TIME COMPLEXITY:

O(Klogn) per testcase + preprocessing to compute $\binom{k}{r}$

PREREQUISITES:

Path Programming, Catalan Numbers.

Problem Statement:

Given two numbers **N** and **K**, find the probability that a **N** length correct bracket sequence can be converted to a good correct bracket sequence in atmax **K** 'good' swaps. A swap is good if the sequence after the swap is a correct bracket sequence.

Explanation:

The problem basically reduces to finding the number of correct bracket sequence having less than or equal to K closing brackets in the first half of the sequence. Finding the number of such correct bracket sequences can be done using Path Programming.
Let the length of the sequence be 2*n. So finding the total number of correct bracket sequences is equivalent to finding the number of paths from (0,0) to (n,n) which does not touch y = x+1 line. Now consider those paths which touch the line y=x+1. If any such path is reflected along the line y=x+1 after the first intersection, it ends at the point (n-1,n+1) which is the reflection of (n,n) about the line. So the number of paths which do not touch the line is equal to **C(2n,n)-C(2n,n-1)**.

Let the number of closing bracket sequences in the first half of the sequence be x. Those paths will pass through (n-x,x). Therefore the number of such paths is equal to **(C(n,x)-C(n,x-1))^2** where 0<=x<=min(k,n/2).So the probability turns out to be **summation(i=0 to min(n/2,k))(C(n,i)-C(n,i-1)))/(C(2n,n)-C(2n,n-1))**.

Also note that ‘good’ swaps does not affect the overall solution since a ‘)’ in the first half of the sequence can be replaced with a ‘(’ in the next half and the sequence will still remain a CBS.

For more details refer to here 5 .

Author Solution 6

PREREQUISITES:

COMBINATORICS, MATH

PROBLEM:

Given an equation $X_1 + X_2 + \ldots + X_N = K$ and an array of N numbers $a_1, a_2, \ldots, a_N$. The task is to find the number of whole number solutions satisfying the given equation such that $X_i >= a_i$ for all i (1<=i<=N). Since the no of solutions can be large, print them modulo $10^9 + 7$.

EXPLANATION:

In this question, we can easily find the case for no solution that if $\sum_{i=1}^{N} X_i$ > K, then no solution exist for that equation. In all other conditions, we can find at least one whole number solution. In order to find the number of whole number solutions for the given equation. Let p = $\sum_{i=1}^{N} X_i$ - K. Now our actual problem reduces to the problem of finding the no of whole number solutions in the new equation, $Y_1 + Y_2 + \ldots + Y_N = p$ without any constraint. The no of solutions to the new equation is similar to the number of ways of distributing p chocolates to N children which can be done in $\binom{p+n-1}{n-1}$ ways.

PREREQUISITES:

Game theory, Combinatorial game theory, Sprague-Grundy theorem

PROBLEM:

A game is played by two players with N integers $A_1, A_2, \ldots, A_N$. On his/her turn, a player selects an integer, divides it by 2, 3, 4, 5 or 6, and then takes the floor. If the integer becomes 0, it is removed. The last player to move wins. Which player wins the game?

QUICK EXPLANATION:

The solution uses the **Sprague-Grundy theorem.** Define $G(n)$ as the following (recursively):

$$G(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2 & \text{if } n \in 2, 3 \\ 3 & \text{if } n \in 4, 5 \\ G(\lfloor n/12 \rfloor) & \text{if } n \geq 6 \end{cases}$$

If the bitwise XOR of $G(A_1), G(A_2), \ldots, G(A_N)$ is 0, then the second player wins (`Derek`), otherwise the first player wins (`Henry`).

EXPLANATION:

This is an example of an impartial game [3] played under normal play condition [1], so the **Sprague-Grundy theorem** [5] applies. The Sprague-Grundy theorem states that every such game is equivalent to a nimber [1]. We can look at each A_i as a game independent of the other values, and the game itself as the *sum* of these N games. So, the theorem implies that every integer A_i can be replaced by a pile of nim of a certain size, and winning positions are preserved, i.e., winning positions are converted into winning positions, and losing positions are converted into losing positions.

To be more specific, we will convert the game into a game of nim with N piles, where the size of the i th pile is the nimber that is equivalent to A_i. Let's call this size $G(A_i)$. The proof of the theorem implies that $G(n)$ is equal to the smallest nonnegative integer that is not equal to $G(n')$ for any n' that is a successor of the game, i.e. $n' \in \{\lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \lfloor n/4 \rfloor, \lfloor n/5 \rfloor, \lfloor n/6 \rfloor\}$. In other words, $G(n)$ is the minimum excluded value of the set $\{G(\lfloor n/2 \rfloor), G(\lfloor n/3 \rfloor), G(\lfloor n/4 \rfloor), G(\lfloor n/5 \rfloor), G(\lfloor n/6 \rfloor)\}$. Since the integer 0 is always removed, we can set $G(0) = 0$.

But recall that a nim position is winning if and only if the **bitwise XOR** of the sizes of the piles is nonzero! Thus, in the original game, the first player is winning if the bitwise XOR of the numbers $G(A_1), G(A_2), \ldots, G(A_N)$ is not zero. Thus, we can answer the problem if we can compute $G(n)$ for any n.

If you're unfamiliar with Sprague-Grundy theorem, we will explain in the appendix why the original game is equivalent to the nim game with pile sizes $G(A_1), G(A_2), \ldots, G(A_N)$.

Computing $G(N)$

We can easily compute $G(N)$ by definition:

```
def G(n):
    if n == 0:
        return 0
    else:
        return mex({G(n/2), G(n/3), G(n/4), G(n/5), G(n/6)})

def mex(s):
    value = 0
    while s contains value:
        value++
    return value
```

Unfortunately, this is very slow, because each call to $G(n)$ needs *five* recursions! This easily blows up quickly, and you'll find that it takes a really long time even for $G(10^8)$. For $G(10^{18})$ there's no hope.

We can optimize this by **memoizing** $G(n)$, that is, storing previously computed values so they only need to be computed once. This solution actually makes computing a single value of G easier; $G(10^{18})$ now finishes instantly! Sadly though, doing this $N = 100$ times for $T = 1000$ cases is still too slow. So how do we compute G much more quickly?

Let's look at the sequence $G(0), G(1), G(2), G(3), \ldots$ first. Here is the sequence:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	2
$G(n)$	0	1	2	2	3	3	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1

To write it more compactly, let's express it as a sequence of **runs**:

n range	$G(n)$
$[0, 1)$	0
$[1, 2)$	1
$[2, 4)$	2
$[4, 6)$	3
$[6, 12)$	0
$[12, 24)$	1
$[24, 48)$	2
$[48, 72)$	3
$[72, 144)$	0
$[144, 288)$	1
$[288, 576)$	2
$[576, 864)$	3
$[864, 1728)$	0
$[1728, 3456)$	1
$[3456, 6912)$	2
$[6912, 10368)$	3
$[10368, 20736)$	0

Where $[x, y)$ denotes the set $\{n \in \mathbb{Z} : x \leq n < y\}$. Notice something interesting if we express the range bounds another way:

n range	$G(n)$
$[0, 1)$	0
$[1, 2)$	1
$[2, 4)$	2
$[4, 6)$	3
$[6, 12)$	0
$[1 \cdot 12, 2 \cdot 12)$	1
$[2 \cdot 12, 4 \cdot 12)$	2
$[4 \cdot 12, 6 \cdot 12)$	3
$[6 \cdot 12, 12 \cdot 12)$	0
$[1 \cdot 12^2, 2 \cdot 12^2)$	1
$[2 \cdot 12^2, 4 \cdot 12^2)$	2
$[4 \cdot 12^2, 6 \cdot 12^2)$	3
$[6 \cdot 12^2, 12 \cdot 12^2)$	0
$[1 \cdot 12^3, 2 \cdot 12^3)$	1
$[2 \cdot 12^3, 4 \cdot 12^3)$	2
$[4 \cdot 12^3, 6 \cdot 12^3)$	3
$[6 \cdot 12^3, 12 \cdot 12^3)$	0

Hmmm. It seems that the sequence $\{G(n)\}$ actually follows a simple pattern! Namely:

- $G(0) = 0$
- If $n \in [12^k, 2 \cdot 12^k)$, then $G(n) = 1$.
- If $n \in [2 \cdot 12^k, 4 \cdot 12^k)$, then $G(n) = 2$.
- If $n \in [4 \cdot 12^k, 6 \cdot 12^k)$, then $G(n) = 3$.
- If $n \in [6 \cdot 12^k, 12 \cdot 12^k)$, then $G(n) = 0$.

But is this true in general? And how do we prove that? Well, it turns out that you can easily prove this by induction:

- For the base case, you can easily compute $G(n)$ for $n < 12$ by hand to see that the statement is correct.
- For the first case, assume $n \in [12^k, 2 \cdot 12^k)$. Then $\lfloor n/2 \rfloor \in [6 \cdot 12^{k-1}, 12 \cdot 12^{k-1})$, and $G(\lfloor n/2 \rfloor) = 0$ by induction, which implies that $G(n) \geq 1$. On the other hand, $G(\lfloor n/j \rfloor)$ cannot be equal to 1 because $\lfloor n/j \rfloor$ is always in the range $[2 \cdot 12^{k-1}, 12 \cdot 12^{k-1})$, and by induction $G(\lfloor n/j \rfloor) \neq 1$. This implies that 1 is the minimum excluded value, and $G(n)$ is actually 1.
- For the second case, assume $n \in [2 \cdot 12^k, 4 \cdot 12^k)$. Then we can similarly show that $G(\lfloor n/2 \rfloor) = 1$ and $G(\lfloor n/4 \rfloor) = 0$, which implies that $G(n) \geq 2$. On the other hand, $G(\lfloor n/j \rfloor)$ cannot be equal to 2 because $\lfloor n/j \rfloor$ is always in the range $[4 \cdot 12^{k-1}, 2 \cdot 12^k)$, and by induction $G(\lfloor n/j \rfloor) \neq 2$. This implies that 2 is the minimum excluded value, and $G(n)$ is actually 2.
- The third and fourth cases can be proven with similar arguments as above.

So now we have another way to compute $G(n)$:

```
def G(n):
    if n == 0:
        return 0
    k = 0
    while 12**(k+1) > n: // '**' is exponentiation
        k += 1
    q = n / 12**k
    if q < 2:
```

```
        return 1
    if q < 4:
        return 2
    if q < 6:
        return 3
    return 0
```

This is much faster than before and is actually fast enough to solve the problem!

We can actually implement $G(n)$ another way by noticing that $G(n) = G(\lfloor n/12 \rfloor)$:

```
G_small = [0,1,2,2,3,3,0,0,0,0,0,0]
def G(n):
    return (if n < 12 then G_small[n] else G(n/12))
```

This has the same running time as the previous $G(n)$ but is easier to type!

The time complexity of computing $G(n)$ is is $O(\log_{12} n) = O(\log n)$, so the running time of the overall algorithm is , or more tightly, O(\sum_{i=1}^N \log A_i).

Appendix: Equivalence to nim

If you're unfamiliar with Sprague-Grundy theorem, here we'll try to explain why the original game with the integers [A_1, A_2, \ldots, A_N] is equivalent to the nim game with pile sizes [G(A_1), G(A_2), \ldots, G(A_N)]. Here, "equivalent" means that the first game is a winning position if and only if the second game is also a winning position.

Recall that in nim, a *move* consists of choosing a pile and removing a nonzero number of things from the pile. You can also look at it as choosing a pile and then *replacing* it with a strictly smaller pile. They're clearly equivalent, but thinking of it using the latter way will make it easier to understand this section.

Suppose we're playing nim with pile sizes [a_1, a_2, \ldots, a_N]. It's a well-known fact that **this is a winning position if and only if a_1 \oplus a_2 \oplus \ldots \oplus a_N is nonzero**, where \oplus denotes bitwise XOR. But why is this true? Here's why.

- If a_1 \oplus a_2 \oplus \ldots \oplus a_N = 0, then any move will make the bitwise XOR nonzero. To see why, suppose your move was to reduce the i th pile from a_i to a_i'. Then the new bitwise XOR will be equal to

\begin{aligned} & a_1 \oplus a_2 \oplus \ldots \oplus a_i' \oplus \ldots \oplus a_N\\ &= (a_1 \oplus a_2 \oplus \ldots \oplus a_i \oplus \ldots \oplus a_N) \oplus a_i \oplus a_i'\\ &= 0 \oplus a_i \oplus a_i'\\ &= a_i \oplus a_i' \end{aligned}

```
which is nonzero because $a_i \not= a_i'$.
```

- If a_1 \oplus a_2 \oplus \ldots \oplus a_N \not= 0, then there exists a move that will make the bitwise XOR equal to 0. To show this, let x = a_1 \oplus a_2 \oplus \ldots \oplus a_N, and let k be the largest 1 bit in x. Thus, there exists an i such that the k th bit of a_i is 1. (Otherwise, the k th bit of x cannot be 0.) The move is to replace a_i with a_i \oplus x. Clearly, doing that move will make the bitwise XOR 0, but it's only a valid nim move if a_i \oplus x < a_i. But this is true because the largest bit where a_i \oplus x and a_i differ is the k th bit, the k th bit of a_i is 1, and the k th bit of a_i \oplus x is 0. Thus, a_i \oplus x < a_i!

Now that we know the strategy for nim, let's now prove why the original game is equivalent to the nim game [G(A_1), G(A_2), \ldots, G(A_N)]. Recall that G(n) = \operatorname{mex}\{G(\lfloor n/2\rfloor), G(\lfloor n/3\rfloor), G(\lfloor n/4\rfloor), G(\lfloor n/5\rfloor), G(\lfloor n/6\rfloor) \}, where \operatorname{mex}(S) is the smallest nonnegative integer not in S.

Clearly, any move in the nim game [G(A_1), G(A_2), \ldots, G(A_N)] corresponds to some move in the original game [A_1, A_2, \ldots, A_N] because of the way G(n) is defined: If G(n) = v, then every value v' < v can be obtained as G(\lfloor n/k\rfloor) for some k\in \{2,3,4,5,6\}. Thus, any move in the nim game can be simulated in the original

game: Replacing G(n) with a smaller value v' is equivalent to dividing n by k so that G(\lfloor n/k\rfloor) = v'. Thus, we can also simulate a “winning strategy” in the nim game in the original game, if such a strategy exists. And if no such strategy exists, i.e., when G(A_1)\oplus G(A_2)\oplus\ldots\oplus G(A_N) = 0, then any move we make will yield a winning position for the next player.

Unfortunately, some moves in the original game actually *increase* the G-value, and such moves don’t have equivalents in the nim game! But that’s no problem, because we can show that such moves don’t matter:

- If you’re in a losing position and perform a move that increases the G-value, then the enemy can respond by replacing that new G-value with the original one. In other words, if G(n) = v and you move so that the G-value becomes v' > v, then the enemy can move so it becomes v again. Such a move exists by our definition of G(n). But we’re now in a position equivalent to the original one!
- If you’re in a winning position, then there’s no point for you in moving so that the G-value increases. Simply follow the strategy in the equivalent nim game, and if the opponent performs a move that increases the G-value, just respond by replacing it back!

Time Complexity:

O(N\log A_\max)

[OFFICIAL] Basic Math/Combinatorics Problems

CodeChef-DSA-learners basic combinatorics math

sidhant007 6 May '20

Hi all,

Let's have discussion about generic questions related to combinatorics, probability, expectation, etc. This can be run in parallel with the current weekly problem sets, since improving in Math is a long process. (Avoid using a computer to try to solve these questions, you should aim to make a closed form formula for these questions)

So I am uploading some questions here, if you have any good questions of similar taste in mind do share them here 🙂

We will first do some easier combinatorics questions, later on we can move to expectation, probability, recurrences, etc.

Incase you think you don't have enough background in probability/combinatorics then refer to this place it is a good resource with good theory + good problems: Brilliant.org discrete math section 797

1a. Given a 2d grid how many ways are there to go from (0,0) (bottom left corner) to (4,6) (top right corner) given you are allowed to move only right and up?
1b. Count the number of ways to go from (0,0) to (5,5) when you allowed to move right and up. But you are also allowed to move one step left at most once ?

Solution

1a. Think of going right as R and going up as U. Then your path is basically a 10 character string of 4Rs and 6Us. Say your string starts with "RUU", then it means first step take a right then take two ups and so on. So how many ways are there to go from (0, 0) to (4, 6)? It is equal to number of ways to re-arrange the 4Rs and 6Us in 10 dashes, i.e C(10, 4).

1b. TBA

2a. Number of palindromes using [a-z] of length exactly 7?
2b. How about for general length = N?
2c. How many recursively palindromic binary strings are there of length N? (A string "s" of length N is recursively palindromic is "s" is a palindrome and the first half of "s", i.e from s[0] to s[N/2 - 1] is also recursively palindrome). For example for N = 5, there are four such strings {00000, 00100, 11011, 11111}, so answer is 4.

Solution

2a. Length is 7 so treat it as 7 dashes. The first 3 dashes fix the characters for the last 3 dashes as well and the middle dash can be anything. So answer is 26^4.

2b. For general N, extend this idea to $26^{\lceil \frac{N}{2} \rceil}$. This is ceiling function, because for odd N, the middle dash provides extra 26 options, but for even there is no such middle dash.

2c. TBA

3a. Number of factors of 840 (including 1 and 840 itself)?
3b. Sum of all the factors of 840 (including 1 and 840 itself)?
3c. Number of factors and sum of factors for a generic N?

Solution

3c. Explaining the generalised version, can extend it for specific N = 840.

Let $N = p_1^{k_1} \times p_2^{k_2} \times \cdots = \prod_{i=1}^{m} p_i^{k_i}$, where each p_i is a unique prime number and $k_i > 0$ and m denotes the number of distinct prime factors of N, i.e this representation is the prime factorisation of N.

Now using this, you can see that the number of factors is $(k_1 + 1) \times (k_2 + 1) \times \cdots = \prod_{i=1}^{m} (k_i + 1)$
Proof: Your factor must be have its prime factorisation as a "subset" of the prime factorisation of N, i.e let K be a factor of N and its prime factorisation be $p_1^{q_1} \times p_2^{q_2} \ldots p_m^{q_m}$, then $q_i \leq k_i, \forall i \in [1, m]$.

May 2020

1 / 20
May 2020

Aug '21

Now each of the prime has to come lesser of equal number of times than it comes in N. So for p_1 how many possibilities do we have? We have $p_1^0, p_1^1, \ldots, p_1^{k_1}$, i.e $(k_1 + 1)$ possibilities. Similarly for a general p_i we have $(k_i + 1)$ possibilities. Since each prime is independent for us, therefore answer is $\prod_{i = 1}^{m}(k_i + 1).

Sum of factors = $(p_1^0 + p_1^1 + p_1^2 + \cdots + p_1^{k_1}) \times (p_2^0 + p_2^1 + \cdots + p_2^{k_2}) \times \cdots = \prod_{i=1}^{m}(\sum_{j=1}^{k_i}(p_i^j)) = \prod_{i=1}^{m} \frac{p_i^{k_i+1}-1}{p_i-1}$

Proof: Think of the polynomial $(p_1^0 + p_1^1 + p_1^2 + \cdots + p_1^{k_1}) \times (p_2^0 + p_2^1 + \cdots + p_2^{k_2}) \times \cdots = \prod_{i=1}^{m}(\sum_{j=1}^{k_i}(p_i^j))$. Then try to actually manually work out each term of this expression. Notice that each term will be of the form $p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_m^{a_m}$, where this term is actually one of the factors of N. Also notice that each factor will come at least once and not be duplicated, i.e it will come exactly once. What this means is that that all the terms of this expression are all the factors. And all the terms are under addition, so they are giving us the same only. That is why the expression is equal to the sum of all the factors (including 1 and N)

4a. Given x + y + z = 10, x >= 0, y >= 0 and z >= 0, how many integer solutions, i.e triplets of integer (x, y, z)'s are there satisfying the above system of equations?
4b. How about x + y + z = 10, x >= 1, y >= 2 and z >= 3. Now how many ?

Solution

4a. Think of forming 10 as 10 stones and x, y, z as 2 sticks. So you have 10 stones and 2 sticks and you want to put these sticks in between these 10 stones which are present in a straight line. x will be equal to the number of stones between the start and the first stick, y will be the number of stones between first and second stick and z will the be the number of stones between the second stick and the end. Notice that solving number of non-negative solutions to x + y + z = 10 is now equal to the number of ways we can put these 2 sticks. So we have 10 + 2 = 12 dashes and we want to put these 2 sticks in between, i.e C(12, 2).

4b. TBA

I hope to add more questions as and when I feel we can move to harder questions.

Feel free to have an attempt at these questions and share you solutions/explanations in the comment section below 🙂

Update 1: I saw a good response and will be working on adding more problems soon. Meanwhile I have added solutions to some of the parts. Please, please dont look at those solutions immediately after reading the question. Also TBA = To Be Announced.

Hello guys,

I am writing about most of the mathematics topics involved in competitive Programming.

GCD (Greatest Common Divisor) 181
→ Properties of GCD
→ Brute Force and Euclidean algorithm to compute GCD
→ Analysis of Euclidean algorithm

Modular Arithmetic 53
→ Basic properties of modulo
→ proof of condition for inverse modulo to exist
->Modulo inverse using extended euclidean algorithm with proof
→ Modulo inverse using Fermat's theorem
→ Solution of Modulo inverse of all integers less than m

Prime Numbers 38
→ Properties of prime
→ Wilson's theorem
→ School method for primality testing
→ Fermat's theorem and its drawbacks
→ Miller-Rabin test with proof
→ Why Miller-Rabin is better
→ Deterministic version of Miller-Rabin
→ Seive of Eratosthenes

Prime Factors 13
→ Properties of Prime Factors
→ Proof of Number of divisors
→ Trial Division Method for Prime Factors
→ Optimization for trial Division method
→ $O(N^{1/3})$ algorithm for calculating number of divisors
→ Logarithmic Prime Factorisation Using Seive

Star and Bars - Combinatorics 23 (**Latest Post**)
→ Star and Bars theorem with proof
→ Number of non-negative integer sum
→ Number of lower bound integer sum
→ Coefficient of x^n in $(1-x)^{-k}$

Combinatorics (coming soon)

Probability (coming soon)

Geometry (coming soon)

Miscellaneous topics (coming soon)

Please tell me if something is wrong or you would like me to add some more information.

Thank you and Happy Coding