Final Standings (standings)

Author: Stefan Dascalescu

Developer: Stefan Dascalescu

Solution

This problem is meant to be an implementation exercise. In short, we will need to be careful about handling the input data, as we will need to store for each problem and each team the amount of penalties they have taken, as well as whether they solved the problem or not. Basically, we need to be a bit careful about storing these pieces of information, while also making sure to avoid overcounting the number of solved problems, as a team can potentially submit multiple solutions for the same problem.

Then in the end, all we need to do is to sort the data as well as applying all the tiebreakers mentioned in the problem statement. Handling all of these things will ensure a linear solution with respect to Q, while also taking in consideration the complexity required to sort the N participating teams, which can be $O(n \log n)$ or $O(n^2)$.

From Constanta to Timisoara (CTTM)

Author: Bogdan-Ioan Popa, Vlad-Mihai Bogdan

Developer: Bogdan-Ioan Popa, Vlad-Mihai Bogdan

Solution

Let $d(a, b)$ = the minimum distance to reach node b starting from node a. For a fixed pair (a, b) the minimum distance of a path from a to b which passes through nodes (X, Y) will be $min(d(a, X) + d(X, Y) + d(Y, b), d(a, Y) + d(Y, X) + d(X, b))$.

We'll calculate dx_a = the minimum distance from node X to node a and dy_a = the minimum distance from node Y to node a using Dijkstra's algorithm. We'll sort the nodes in increasing order with respect to the value $dy_a - dx_a$. Now we will iterate through the nodes. With a fixed node a we want to know how much it contributes to the total sum. a will go through x when pairing it with a node b which appears earlier in the sorting, and it will go through y when pairing it with a node b which appears later in the sorting. The time complexity of the solution is $O((M + N)log(N))$

Count Critical Edges (cntcrit)

Author: Bogdan-Ioan Popa

Developer: Vlad-Mihai Bogdan, Bogdan-Ioan Popa

Solution

Every edge will contribute to the total sum an equal number of times. So we will focus on counting how many times does edge $(1, 2)$ appear as a critical edge. If edge $(1, 2)$ is critical, it means that if removing it node 1 and 2 will be in two different connected components. So we will iterate through the size of the connected component of 1, count how many connected graphs are there with the given size, and the rest of the graph can be anything. So now the problem reduces to counting how many labeled connected graphs with N nodes exist.

Let $dp_N =$ the number of labeled connected graphs with N nodes.

$$dp_N = 2^{\binom{N}{2}} - \sum_{j=0}^{N-2} dp_{j+1} \cdot \binom{N-1}{j} \cdot 2^{\binom{N-j-1}{2}}$$

The j in the recurrence iterates through the size of the connected component of node 1.

Chalk Board (chalkboard)

Author: Ovidiu Rata

Developer: Stefan Dascalescu

Solution

The *Fundamental theorem of arithmetic*, that states that a number $x = {p_1}^{a_1} \times {p_2}^{a_2} \times \cdots \times {p_n}^{a_n}$, where $p_1, p_2, \ldots, p_n$ are prime factors and $a_1, a_2, \ldots, a_n$ their powers then the number of divisors is equal to $(a_1 + 1) \times (a_2 + 1) \times \cdots \times (a_n + 1)$.

In order to solve the problem, we can first precompute the prime factorization of the integers from 1 to 10^6 either by running a sieve at the beginning, or doing trial division for each of the q numbers, that is for the numbers we get for the operations of types 1 and 2.

Regardless of the way we use for handling the prime factorizations for the first two types of queries, we can use a segment tree which handles point updates on one of the values from 2 to 10^6, as well as queries which compute the number of possible integers we can obtain using primes within a certain range. The most important thing is that for a prime number p, if the prime factorization of the number written on the chalk board contains x^y, where y is the exponent of the prime factorization of x, we can create $y + 1$ such numbers. Then, for two or multiple prime factors, we can multiply these answers and we will use this in the logic of computing the products on the segment tree, while making sure to avoid overflows and other such issues.

The final complexity of the algorithm will be $O(q \cdot mx \cdot logVALMAX)$, where mx is the highest number of distinct prime factors a value has, and $VALMAX$ is the upper bound for the values of the input, which is 10^6.

Connected (connected)

Author: Alexandru Lorintz

Developer: Alexandru Lorintz

Solution

For the first subtask, a $O(N^3)$ solution where a range from the permutation is fixed (this step has time complexity $O(N^2)$) and a traversal algorithm is used to check the connectivity (this step has time complexity $O(N)$) is enough to solve it.

In order to optimize the previous solution, one can make use of a **Disjoint Sets Union** data structure in order to update the connectivity state of a range when it is extended, improving the time complexity to $O(N^2 * \alpha(N))$, which is enough to solve the second subtask.

In order to solve the full problem we need to make some observations regarding the problem. The first and most important one I would say is that a connected region from a given tree will also always be a tree by itself, so a way of characterising its structure is using the mathematical relationship between the number of nodes and edges of a tree, i.e. $V = E + 1$, where V represents the number of nodes and E the number of edges. Furthermore, I will say that any set of nodes from a tree for which this equality holds (the edges considered will be the edges that have both ends in nodes from the given set) is a connected region of that tree, as it is easy to visualize that if the set of nodes form a forrest of trees, the difference $V - E$ is always **greater** than 1.

Having the previous observations in mind, we can actually describe a continuous subsequence of nodes from the permutation by the difference $V - E$, which must be 1 in order for that subsequence to be a connected region and will be greater than 1 otherwise.

Let's consider a data structure where information for all the continuous subsequences that end in some index is kept. Formally, if the current ending index is R and our data structure is DS, we will consider $DS[L] = V[L] - E[L]$ for all values $L \leq R$, where $V[L]$ is the number of nodes from the range $[L, R]$ (which is just $R - L + 1$) and $E[L]$ is the number of edges induced by these nodes.

In order to find out how many subsequence that end in R are connected, we need to count the number of values L such that $DS[L] = 1$ (as stated above). After that, R needs to be gradually increased ($R = R + 1$) until we reach the end of the permutation and at each step perform the desired query (we obviously start with $R = 1$). This means that the data structure must be updated for the new ranges.

Each update will be split into observing how changes are made to $V[L]$ and $E[L]$. The update for $V[L]$ is quite standard, as the only thing that happens is the appearence of a new node (corresponding to the index $R+1$), so all values $DS[L]$ with $L \leq R$ are increased by 1 and $DS[R+1]$ is set to 1.

For updates regarding values $E[L]$, new edges having an endpoint in the node with index $R+1$ must be considered. If there is a new edge having one endpoint in the node with index $R+1$ and the other in a node with index $L \leq R$, this edge updates the state of the current relevant values from the data structure for all the subsequences $[P, R+1]$ with $P \leq L$, so all values $DS[P]$ need to be decreased by 1 (because edges contribute to DS with negative value).

Now that the details of what the data structure should be capable of are mentioned, it should be clear that a suitable candidate for the job would be a segment tree that supports lazy update for increasing all values from a range with some given value and query for computing the minimum value from a range and its frequency (as it was mentioned earlier that $V - E \geq 1$, so if 1 appears on a range it will also be the minimum and we are only interested in this).

With all of this said, the time complexity for solving the whole problem becomes $O(N * logN)$ with $O(N)$ space complexity.

Mermaid of the Waters (`mermaid`)

Author: Vlad-Mihai Bogdan

Developer: Vlad-Mihai Bogdan

Solution

For a given binary string S, we can use a stack to check if the string can become empty using the operation. The greedy solution would just be to erase the i^{th} character of the string if the character on top of the stack is the same as S_i.

If we look at the stack (at any point of time), we can see that the it's values alternate (we either have $010101\ldots$ or $101010\ldots$). So, this could lead us to an $O(N^2)$ dynamic programming solution, where we keep the size of the stack and the first character inserted in it.

For the $O(N)$ solution, we can make a bijection between the binary strings of length N where the number of 0's is equal to the number of 1's and the binary strings of length N satisfying the condition stated in the statement.

Another solution could use the fact that the recurrence of the $O(N^2)$ solution builds Pascal's Triangle and one can see the fact that the answer will always be $\binom{N}{N/2}$.

XYQueries (`xyqueries`)

Author: Bogdan-Ioan Popa

Developer: Bogdan-Ioan Popa

Solution

Let $F(x, y)$ be the number of pairs (l, r) where $x \le min(l, r)$ and $max(l, r) \le y$. Then the answer for a query (X_i, Y_i) will be $F(X_i, Y_i) - F(X_i + 1, Y_i) - F(X_i, Y_i - 1) + F(X_i + 1, Y_i - 1)$. So we will transform each query into 4 queries of the F function. For solving these queries we will use MO's algorithm but we have to keep the frequencies of the values in mind. We will sort the pairs (A_i, i) in lexicographic order. Now a query of the F function will be a range query in the sorted array of pairs. For a query we will 'activate' the positions of the values and we will count the number of subarrays which have all the elements activated. To do this efficiently we will have to solve these queries only by activating positions. So the queries of length less than $sqrt(N)$ will be solved by brute force. Now the queries of length greater than $sqrt(N)$ can be solved only by activating positions (The x and y values will be in different buckets) using MO's algorithm with undo operations (for a query we will have to undo the activations that have been made from x to the end of its bucket). The total complexity of the solution is $((N + Q) * sqrt(N))$

GcdMatrix (gcdmatrix)

Author: Alexandru Lorintz

Developer: Alexandru Lorintz

Solution

For the first subtask, a simple brute-force approach that iterates through all the submatrices of the given matrix with time complexity $O(N^6)$ is enough.

For the second subtask, an optimisation of the previous solution is required. This can be achieved by computing a two-dimensional sparse table that keeps track of the **greatest common divisor (gcd)**. This can be built in $O(N^2 * log^2(N))$ time and space complexity and then queried in $O(1)$ with some pre-proccesing for the value of the **gcd** between any possible pair of values, as the number of them is quite small. Using this data structure it is only required to fix each submatrix and there are only $O(N^4)$ of them, so this becomes the time complexity. A solution with time complexity $O(N^4 * log)$ that does not do the pre-processing for the **gcd** of all pairs of values should also get a full score for this subtask.

For the final subtasks we need a better way of understanding the problem and model a different solution. Let's start with the set of all submatrices as a candidate for the solution. Some of these do not have **gcd** equal to 1, so they should be removed. Let's imagine that we fix submatrices where all the elements are divisible by some value increasingly (this means that their **gcd** is **divisible** by that value). We should subtract all the matrices that have that value equal to 2 and 3 for example, but we should be careful of what happens next. With some careful consideration, we can see that 4 should be ignored because all of these matrices are included **only** in those that have the **gcd** divisibile by 2. For value 5 it's the same case as for 2 and 3, but for 6 we should actually **add** these back to the solution, because they were subtracted twice for 2 and 3, as $6 = 2 * 3$.

With the previous intuition in mind, we can easily get to a solution based on the **Principle of Inclusion and Exclusion** for numbers and their prime factors. We should also only consider the square-free values not greater than the maximum possible value. After this step, if we do as described before, the problem becomes counting the number of submatrices full of **ones** in a binary matrix (we put a 1 if an element from the matrix is divisible by the considered value). This step can be easily solved with a **monotonic stack** approach, similar to the largest rectangle in a histogram problem.