Problem A
AB to C

Author: Shreyan Ray

Let n_A, n_B and n_C denote the number of A, B and Cs in the string. The following observations can be generalized to all pairs of characters and nost just `AB`.
Invariant 1 : The parity of $(n_A + n_B)$ is invariant.
Monovariant 1 : The quantity $(n_A + n_B)$ is non-increasing.
Observation 1 : It is possible to delete `BB` by using a `A`, i.e. convert `ABB` to `A`. First convert `AB` to `C` and then `CB` back to `A`.

First of all, if S contains all of the same characters, no operation is possible.
Now, assume that S contains at least 2 distinct characters.

Observation 2 : If $(n_B + n_C)$ is even, we can remove all `B` and `C`s from the string.
Proof : If both n_B and n_c are even, simply use Observation 1 to delete all. Otherwise, both are odd, combine 1 of each into an `A`, and then we go back into both even case.

Also, let's note that we can reduce the number of `A`s in the string by first doing operations with `BAA` or `CAA` deleting 2 `A`s at a time, as in Observation 1. Thus, the final string will be either `A` or `AA`. Due to invariant 1, it is unique, and this is the optimal answer for the case (n_B+n_C) is even.

Now, we come to the case when $(n_B + n_C)$ is odd. Here, again by invariant 1, we will come to the conclusion that not all `B` or `C`s can be deletd from the string. However, clearly the final string should satisfy $n_B + n_C = 1$, otherwise we can either delete 2 `B`, or delete 2`C`, or combine `BC` to form `A`.

Further, we can note that not all strings with $(n_B + n_C) = 1$ are optimal either. The idea is that if we have a `B` or `C` in the middle of the string, we can delete it by combining it with a `A`, and send the result to the back of the string. This actually gives a better result. For example, compare `ABAA` and `AAC`.

With this idea in mind, we can note that the final string has 3 possible optimal endings, `B`, `C` or `BA`; preceded by a prefix of `A`s. To minimize lexicographically, we have to maximize the size of the prefix of `A`s.

Here, we use Monovariant 1 to note an upper bound of $n_A + min(n_B, n_C)$ on the number of `A`s possible, and infact we can combine $min(n_B, n_C)$ `BC` to generate these many `A`s. Then, we can choose the leftover `B` or `C`, and delete them 2 at a time by using a `A` as in Observation 1.

We have achieved the upper bound on the number of `A`s, however it may happen that not all `A`s are on the prefix, and some are after the `B` or `C`. Here, a simple greedy idea is provably correct, that is where we try to send as many `A`s to the front, and keep the last `B` or `C` as late as possible. Note that even while deleting `CC`, 1 `A` can be sent to the front of the string.

If the string still does not end with `B`, `C` or `BA`, we will combine the last remaining `B` or `C` with a `A` and send the result to the back of the string.

The whole process can be implemented in $O(N)$.

Solve Count: 17.
First Solve: segment trACk at `1:01`

Problem B
Bob Learns To Read

Author: Shreyan Ray

If Bob had only 1 word A, the answer is simply $d-1$ where d represents the number of distinct characters in A. This is because we can always group up equal characters together to lower the difficulty for Bob.

Similarly, with 2 strings, we can easily obtain $d_A + d_B - 1$ as the difficulty level by grouping equal characters in the 2 strings.

Further, if the 2 strings shared some character, we can make sure our rearranged string A ended with that character, and our string B started with it. This gets us a difficulty level of $d_A + d_B - 2$.

It can be proven that this is the optimal answer. We can find d_A and d_B by maintaining a boolean array of whether each character appeared in A or not. Thus, the problem is solved in $O(|A| + |B| + 26)$.

Solve Count: 113.
First Solve: Chota Haathi at `0:05`

Problem C
Check Good

Author: Satyam Kumar Roy

Let's start with characterizing *good* sequences.
Here are some necessary conditions for a sequence $B_1, B_2, ..., B_M$ to be good (assume that B is sorted) :

- $B_1 = 0$
- $B_M = (M - 1)$
- $B_i \geq (i - 1)$

It's not hard to see that the following conditions are also necessary, because we can construct a permutation P corresponding to it.

Now, some preliminary work. We can note that the third condition basically states that there are at most i elements $< i$. For each L, we can find $maxR_l$ as the right most index where this is true. This can be done using a lazy segment tree and 2 pointers. Also, we can find $minR_L$ as the leftmost index x where $[L, x]$ contains a 0.

Now, we need to check intervals $[L, R]$ such that $minR_L \leq R \leq maxR_L$ and these 2 constraints automatically take care of the 1st and 3rd constraints in the characterization. So, we only care about the 2nd constraint.

There are a few approaches one can take now.

Approach 1

Let's fix the value of B_M, i,e. fix the maximum element. Note that this also fixes the length of the interval under consideration.

Let $[l, r]$ be the range where M is maximum in. Then, we can actually bruteforce the subarrays of length exactly $(B_M + 1)$, wholly enclosed in $[L, R]$ and containing i.

This is because this will run in $O(min(r - i, i - l))$, i.e. smaller of the left and right halve of the maxima interval. And, it is known that this sums to $O(Nlog(N))$.

Each interval can be quickly checked in $O(1)$ using the precomputed arrays. Thus, the solution runs in $O(Nlog(N))$.

Approach 2

Fix the left end of intervals under consideration to be L. We get a subarray to check in. We only care about the elements which are prefix maximas from L.

For each index i, we can also generate $minL_i$ and $maxL_i$, where this denotes the minimum and maximum indices of L such that $[L, R]$ can be good for some $R \geq i$, and i being the maxima element.

After this, a sweepline and maintaining good indices solves the problem. The details are slightly complicated, you can refer to the jury code. This is also $O(Nlog(N))$

Solve Count: 1.
First Solve: SubtasksWhere at `4:13`

Problem D
Hard Counting Problem

Author: Shreyan Ray, Satyam Kumar Roy.

Observation : $f(A) \leq 2$
Proof : Suppose one round of deleting elements happened and the minimum index of a deleted element is g. Then, all undeleted indices satisfying $i > g$ will get deleted on the next turn, while indices $1...(g-1)$ always remain stable.
This is because since i was not a deleted index, $A_i = i$ is true. But, now A_i goes to some other position due to being reindexed, say p and $A_i = p$ is impossible as $p \neq i$.
Thus, we can see that all undeleted indices $> g$ get deleted on the 2nd turn, where the array becomes stable.

We need to separately count the number of arrays such that $f(A) = 0, 1$ and 2.

$f(A) = 0$ is pretty easy, since all $A_i = i$ must hold. It is 1 if $n \leq m$, otherwise 0.

To count arrays such that $f(A) = 1$, we require that after the first round of deleting, there is no more deleting. This means that all indices $\geq g$ get deleted on the first turn, since if some index was left, it will get deleted on second turn.

Let's fix the value of g and try to compute the number of arrays. $A_i = i$ must hold for all $i < g$, so there is at most 1 way for this (unless m is too small, where 0). For all $i \geq g$, $A_i \neq i$ holds. For each index $i \geq g$, there are either $M-1$ choices, or M choices depending on $i \leq M$ or not. The number of indices which have $M-1$ or M choices can be computed easily, and then we can use binary expontiation to count the number of arrays.

Add the answer over all g to get the total number of arrays.

$f(A) = 2$ arrays can be found by subtracting the former 2 cases from M^N.

Thus, the problem is solved in $O(N \cdot log(N))$ or $O(N)$ with precomputation.

Solve Count: 85.
First Solve: Fast and Fourier at `0:47`

Problem E
Largest K

Author: Satyam Kumar Roy

Let's use strings in increasing order of their lengths. Whenever we choose S, we can also choose $inv(S)$, where $inv(S)$ represents S but each bit is flipped. This uses an equal number of 0s and 1s, and thus we cannot do better in the case when we pick even number of final strings.

We also need to check if we can fit in one extra string at the end. If the length is odd, the best we can do is use a string with an "imbalance" of 1, and thus requiring remaining $n \geq \lceil \frac{len}{2} \rceil$, while for an even length, we can use a string with imbalance 0 thus only requiring $n \geq \frac{n}{2}$.

To implement this solution, first iterate over the lengths from 1. Check if we can take all strings of this length. If yes, go the next length, otherwise find exactly how many strings we can take of this length and terminate the process. Here, we will first find how many paired up strings we can take, which is simple to calculate since each pair costs us len 0s and 1s, and then check if the leftover is sufficient for 1 extra string or not.

The whole solution can be implemented in $O(log(n))$ per test case.

Solve Count: 101.
First Solve: Fast and Fourier at `0:20`

Problem F
Lexicographic Raffle

Author: Matthew Roh

$X = S[L, R-1], Y = S[L+1, R]$. Let us try to see when X is lex smaller than Y and vice versa.
If $S_L < S_{L+1}$, we can easily see X is lex-smaller; and if $S_L > S_{L+1}$, we can easily see Y is lex-smaller.
This leaves out the case $S_L = S_{L+1}$. Let us then compare S_{L+1} and S_{L+2}. Again, we get the same conclusion that unless $S_{L+1} = S_{L+2}$, we can tell whether X is lex-smaller or not.

Generalizing the idea, we can see that we care about the first index i such that $S_i \neq S_{i+1}$ and $L \leq i < R$. If $S_i < S_{i+1}$, X is lex-smaller, otherwise Y is lex-smaller.
Also, note that if no such index i exists, $X = Y$.

Now, assume that $X < Y$ is true. This means that R decreases by 1 and the process is repeated. However, you can note that R decreasing by 1 does not really change the lex-ordering of X and Y, unless $R = (i+1)$. This is because we only cared about the first index i such that $S_i \neq S_{i+1}$, and that will remain constant unless $R = (i+1)$ mentioned above. This means that if R has decreased on the previous step, it will also decrease on this step (even when $R = (i+1)$, this is true because then $X = Y$).

Thus, once we get a $R--$, the entire process is forced to be $R--$. Consequently, before that point, we can only have $L++$. Hence, our problem is reduced to finding the find point i such that $S[i, R-1] \leq S[i+1, R]$ and $i \geq L$.

Let nx_i denote the first point $> i$ such that $S_i \neq S_{nx_i}$. Then, $S[i, R-1] \leq S[i+1, R]$ if and only if $S_i < S_{nx_i}$.
We can precompute a good array where $good_i = true$ if and only if $S_i < S_{nx_i}$, and then just do a lower bound on a vector of good indices with respect to L to find the point.

Be careful to handle the edge cases of nx_i not existing, and there being no good index in the range $[L, R]$.

Overall, the problem can be solved in $O(Nlog(N))$ or $O(N)$ depending on implementation.

Solve Count: 63.
First Solve: assert(rand()) at 0:57

Problem G
Majority Voters

Author: Shreyan Ray

Let us represent an A by $+1$, and B by -1. Now, A wins if and only if $sum > 0$. Changing a person to a majority voter can change the sum by ± 2. Since we want to minimize the number of people we change, let us try to see when the sum can be changed by 2.

Suppose there exists some prefix sum which is > 0, then, we can take the first B index after this position and turn that to a majority voter. This changes the sum by 2, and also preserves the property of having a prefix sum > 0, thus making us able to do this operation multiple times. Note that eventually, sum will become positive by doing this. Thus, the answer is easy to compute here.
Conversely, it is easy to see that if all prefix sums ≤ 0, it is impossible to obtain a change by 2.

Now, we come to the case where all prefix sums ≤ 0. Anybody who we make a majority voter will not vote for Alice (at least immediately), but will just stop voting for Bob (thus changing the sum by 1).
This continues till we get a prefix sum > 0, after which we can use the changing by 2 operation.

While doing the changing by 1 operation, it is optimal to make the first B index a majority voter each time, as that changes all the prefix sums succeeding it. We need to calculate the number of operations till we get at least 1 prefix sum > 0.

Note that the maximum prefix sum in the range $[L, R]$ will be the first to become > 0, and thus, we just need a structure which helps us compute the maximum prefix sum, and the whole subarray sum. These structures are segment tree and prefix sum respectively.

There is a minor implementation detail that was covered in the samples. The maximum prefix sum should be found in the range $[nx_L, R]$ instead of $[L, R]$, where nx_L denotes the position of the next A.

The total time complexity is $O((N + Q)logN)$.

Solve Count: 34.
First Solve: HeheKyaHua at `0:47`

Problem H
P to Q

Author: Satyam Kumar Roy

A lower bound on the score is clearly $max(inversions(P), inversions(Q))$. Infact, this is achievable!

Let us first convert P to I while making sure all intermediate states have $\leq inversions(P)$, and then I to Q, again making sure all intermediate states have $\leq inversions(Q)$.

To convert P to I, we can choose n and send it to the n-th place, and induct with $n - 1$. This only lowers inversions so it is fine.

To convert I to Q, we can choose Q_n and send it to the n-th place, and induct with $n - 1$. This only creates inversions which are also present in Q and hence it is optimal as well. (Note : you can think of it as reversing the operations used in the first step)

The problem can thus be solved in $O(N)$.

Solve Count: 94.
First Solve: 404_solution_not_found at `0:46`

Problem I
Prime Difference Graph

Author: Shreyan Ray

Observe that $[1, 2, ..., n]$ which is the minimum sum possible is actually valid array for $n \geq 4$. This is because all even numbers, and all odd numbers, are connected because 2 is prime and thus there is an edge $(i, i + 2)$. Further, for $n \geq 4$, there is a $(1, 4)$ edge between the even and odd numbers as 3 is a prime. This makes the graph connected.

Hence, we can output $[1, 2, ...n]$ for $n \geq 4$.

For $n = 1$ and $n = 2$, the output is given in the samples. For $n = 3$, we can see that the optimal array is $[1, 2, 4]$.

Solve Count: 113.
First Solve: 404_solution_not_found at `0:12`

Problem J
Score Sum

Author: Satyam Kumar Roy

First, let's observe that $f(1, n)$ is maximum. This is because, smaller ranges decrease $(R-L+1)$ by 1 and decrease $d(L, R)$ by at most 1. Thus, the value either increases or stays constant.

If all elements are distinct, then $L = R$ or a length 1 sequence is optimal, as all subarrays have the same value of $f(L, R)$.

Otherwise, consider all elements are not distinct. Then, any repeated element cannot be removed from the optimal subarray without reducing the f value. This means that we will delete a maximal length prefix and suffix, such that those values were not present elsewhere in the array (to lower the length of the subarray), and keep the remaining subarray.

Thus, we can now find $score(A)$, but we need the sum over all subarrays.

Let us ignore the subarrays where all elements are distinct for now. They are easy to compute. Let us focus on the maximum prefix which can be deleted for each subarray $[L, R]$ and similarly, the maximum suffix.

Approach 1 :

Let nx_i denote the next occurrence of A_i. Then, an interval $[L, j]$ can be deleted iff $nx_i > R$ for all $L \leq i \leq j$.

Let us inverse what we compute, let us fix $[L, j]$ as the deleted interval, and compute how many valid R exist. Clearly, $R > max(nx_i)$ for $L \leq i \leq j$. Consider the prefix maximas of nx starting from L. A simple math formula can be generated for these indices representing their contribution.

We can then maintain a stack sweeplining backwards, and keep the contributions of each element stored.

The problem can be solved in $O(N)$.

Approach 2 :

Fix R instead. Call an index i good iff $nx_i > R$. Consider the ranges of good elements. A range of length L will contribute $\frac{L \cdot (L+1)}{2}$.

We can sweepline backwards and keep track of the segments with a union find data structure. The solution complexity is $O(N\alpha(n))$.

Solve Count: 1.
First Solve: SubtasksWhere at `4:49`

Problem K
Stalin Sort

Author: Shreyan Ray

There is a core observation here, that seems obvious but is sufficient to solve the problem.
The last element to be deleted must have caused an inversion in the sequence, and other elements of the sequence must be increasing.

Clearly, this is necessary as well as sufficient for a sequence of operations to be valid.

Note that we can model this problem as choosing a random permutation out of the $N!$ permutations, and then deleting in that order, till the array becomes sorted.
Now, there are multiple ways to solve the problem. Let's discuss the model solution.

Let us fix the length of the increasing sequence k, and the number of possible inversion positions l. Then, the number of permutations corresponding to this sequence is simply $k! \cdot l \cdot (n-k-1)!$.

To calculate the number of such sequences, we can do a dynamic programming, $dp[i][k][l] =$ number of increasing sequences ending at i with length k and l possible inversions. Computing this dp array is fairly easy as we can run a for loop on the previous index j, and then compute the extra possible inversions in the range (j, i) using a for-loop.

This gives us a time complexity of $O(N^5)$ but it is easy to optimize to $O(N^4)$. It is also possible to solve the problem in faster complexities, but it was not required.

Solve Count: 12.
First Solve: TrieHarder at `0:22`

Problem L
Yet Another MST Problem

Author: Satyam Kumar Roy

Consider Kruskal's algorithm for MST. We need to find the minimum weight edge connected 2 components.

Let us consider a multisource BFS approach. Run multisource BFS from the marked nodes, and then when you get 2 different distances from 2 different marked nodes, we unite them to the same component.

It can be shown that all the possible edges are generated here, and **almost** in increasing order.

The small edge case is that the distances may be 1 off from sorted order. This was covered in the last few sample tests. The fix is to generate the edges but not unite immediately. Then, unite later on by sorting all the edges.

The time complexity is $O((N + M)log(N))$.

Solve Count: 8.
First Solve: poocha kya at `1:06`