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 $O(N)$.

Problem B
Bin Packing Problem

Author: Shreyan Ray

Observation 1: It is always optimal to match x with $12 - x$.
Proof 1: Assume we put $a_1, a_2, .., a_k$ in the bin with x instead of $12-x$. Clearly, $a_1 + a_2 + ... + a_k \leq 12 - x$. There is some other bin where we put $12 - x$. We can simply swap the positions of these k items and the $12 - x$. This matches x with $12 - x$ without increasing the number of bins used.

Now, let f_i denote the frequency array.
We can assume $f_{12} = 0$, because otherwise we can just output $\texttt{ans} + f_{12}$, where `ans` is calculated assuming $f_{12} = 0$. 12 must always form it's own bin.

While $\min(f_x, f_{12-x}) \geq 1$, we can pair x and $12 - x$ to reduce both by 1 and number of used bins by 1. In the end, either $f_x > 0$ or $f_{12-x} > 0$. Similarly, we can do this with 6 too while $f_6 \geq 2$, so we may assume $f_6 \leq 1$.

Now, we note that there are at most 6 non-zero entries in f. More specifically, we can even assume 5 non-zero entries by bruteforcing on what the last 6 (if it is present) will be paired with.

Let us optimize further by making $f_1 = 0$ and $f_{11} = 0$.
If $f_{11} > 0$, $f_1 = 0$, and so 11 will always be in it's own bin. Thus, this case is easy.
If $f_1 > 0$, let us assume $f_1 = 0$ and we computed the answer to be X. Then $\max(X, \lceil \frac{\sum(A_i)}{12} \rceil)$ is the answer. This is because 1 can fill up any gaps leftover, and we need extra bins if and only if the $\sum(A_i)$ is too large.

Thus, there are only 4 non-zero states. We now write $\texttt{DP}[f_x][f_y][f_z][f_w]$ be the minimum number of bins needed to accomodate f_x number of x, f_y number of y, f_z number of z and f_w number of w.

We can write transitions in a bruteforce manner, which increases our constant factor (but still passes very comfortably). The model generates all tuples (g_x, g_y, g_z, g_w) such that $g_x \ x$, $g_y \ y$, $g_z \ z$ and $g_w \ w$ can fit into 1 bin.

At first sight, this seems $O(N^4)$, however we need to note that each f_i here is expected to be small. Why? It is because of how f was created $f_i = |F_i - F_{12-i}|$, where F_i denotes the initial occurrences of i in the input. In random input, this value is expected to be small.

Formally, it can be proven that each f_i is $O(\sqrt{\frac{N}{6}})$ with probably close to 1 using Bernstein Inequalities to bound. The final number of states becomes $T \cdot \frac{N^2}{36}$ with a non-trivial constant factor due to the number of transitions. Nevertheless, the model runs in 0.15s without any constant factor optimizations.

Problem C
Construct uwu

Author: Matthew Roh

We claim that the provable lower bound is **nearly always achievable**. (Specifically the only counter-example is $n = 5$)

The lower bound is found by finding the smallest string with number of `uwu` subsequences $\geq N$ instead of exactly N. This is relatively standard. Clearly, it is best to have all `w` together, and the number of subsequences is abc where a is number of prefix `u`, b number of middle `w` and c number of middle `u`. For an optimal string, it is not hard to prove that $\max(a, b, c) - \min(a, b, c) \leq 1$ is required, and after that we can compute the shortest string possible in $O(N^{\frac{1}{3}})$ with bruteforce.

To even start constructing, we need to reformulate the problem. Our aim is to get a construction that works for large enough N, while for small N, we can do various things such as a bruteforce DP.

Let's try to make exactly N `uwu` subsequences with a `u` and b `w`. For convenience, we will assume a is even, and $a \approx 2 \cdot b$. For a fixed length L, we can choose a and b depending on mod 3. Obviously, $(\frac{a}{2})^2 \cdot b \geq n$ should hold. Let's start with the string with maximal subsequences, i.e. `uuuu....www.....uuu....`, the block lengths being $\frac{a}{2}, b, \frac{a}{2}$.

Suppose we shifted the last `w` by i units to the right, i.e. we form the string `uuu....www.....uuu....wuuu....` where the block lengths are $\frac{a}{2}, b-1, i, 1, \frac{a}{2}-i$. It is not hard to see that number of `uwu` subsequences reduced by exactly i^2.

Suppose we shifted the second last `w` by j units now, (where shift is counted by number of `w` it overtakes). Then, again the number of subsequences reduces by j^2. And, we can generalize this to any number of `w` we would like to shift.

Let $k = (\frac{a}{2})^2 \cdot b - N$, and let s_i be the amount we shift the i-th `w`. Then, we can convert the problem to the following knapsack problem:
Find array s of length b such that $\sum s_i^2 = k$ and $0 \leq s_i \leq \frac{a}{2}$

Now, note that $a, b \approx O(n^{\frac{1}{3}})$ and $k \approx O(n^{\frac{2}{3}})$. The second expression comes from the fact that we are solving for the minimum length L which means that $(a-1)^3 < N$ while $(a+1)^3 \geq N$, and $k \leq (a+1)^3 - (a-1)^3$ definitely.

This means that the knapsack weights are $O(n^{\frac{2}{3}})$, same as the required target. We can use a greedy strategy of taking the largest perfect square $\leq k$ each time, and it will work out. The reason is that there is **nearly** always a perfect square between X and $2 \cdot X$, and so the number of needed steps is $\frac{2}{3}log(n) + C$ which is definitely $\leq b$ for large enough N.

Formally, with some analysis, it can be proven that the algorithm works for all $n \geq 1000$, even if that is a very conservative estimate. It only actually fails for $n = 5, 15, 20, 41$ and one can either do a dp or manually solve for those cases.

All steps take $O($`output length`$)$ which is $O(N^{\frac{1}{3}})$.

Problem D
Counting Distance Arrays

Author: Shreyan Ray

Claim 1: Subsets S with $|S| \geq 3$ are redundant.

Proof 1: In a tree, it is a well known fact that one of the maximum distant nodes from any fixed node u is always an endpoint of a diameter.

Consider the subtree generated where only the elements of S are leaves. Let this be T'. Then in this tree T', the diameter always has endpoints in S, and the maximum distance of any vertex from any other vertex is to the endpoints of the diameter. Thus, for the subtree, only the 2 elements of S that actually form the diameter matter.

For the nodes outside the subtree, say u, consider the nearest node present in T', let it be x. Then it is not hard to see $a_u = a_x + dist(u, x)$, and thus again only the nodes at the endpoint of the diameter of the subtree matters for all the values. Thus, subsets of size ≥ 3 are redundant.

Among subsets of size 2, some arrays might be formed by multiple subsets.

The idea here is that if your set is (u, v), then the 2 quantities that actually matter are centre of $path(u, v)$ and $d(u, v)$. It can be proven by the observation that $max(d(x, u), d(x, v)) = d(C, x) + d(u, v)$ where C is the centre of path (u, v).

Now, fix C the centre, and count the number of possibilities of $d(u, v)$. (Note that C may be a node, or the midpoint of an edge)

We require that (u, v) pairs actually have C as a centre, so they must be equidistant from C and in different subtrees. Find the largest distance from C in all of the subtrees of the adjacent nodes to C. Then the maximum possible value of $d(u, v)$ will be 2^{nd} max in this list.

We can find this list through a standard rerooting DP for a $O(N)$ solution.

Problem E
Easy Counting Problem

Author: Shreyan Ray

First of all, let's solve for $M \geq N$. Surprisingly this case is much easier than the general problem.

Observeration 1: An index i gets deleted if and only if $A_i \in [i-d, i]$ where d is the number of deleted elements in indices $< i$.

Proof 1: This fairly trivial with a proof by induction. Either $A_i = i$ and it gets deleted eventually (after greater indices being deleted, which don't affect this index), or $A_i < i$ and then something before i will get deleted eventually since $i - d \leq A_i$ decreasing the index at which A_i exists and thus, we can induct on the value of $i - A_i$.

Observation 2: The reason why $M \geq N$ is much easier than general M is that $[i-d, i]$ is always a subset of $[1, M]$, and thus the probability that i is deleted is simply $\frac{d+1}{M}$.

We will now use Linearity of Expectation.
The term $\frac{d+1}{M}$ for index i can be written as $\frac{1 + [B_1] + B_2] + ... + [B_{i-1}]}{M}$, where $[B_j]$ is an indicator variable, which equals 1 if j got deleted eventually, and 0 otherwise.

Now, we can transform this to an expected value equation. The expected value of B_i, say E_i is simply $\frac{E_1 + E_2 + ... + E_{i-1} + 1}{M}$. You can note that E_i is simply the probability i gets deleted, by definition of B_i and E_i.

Thus, this is a simple linear dp to find the expected number of deletions in $O(N)$ for $M \geq N$.

We now extend to $N - 20 \leq M < N, N \leq 10^5$. We will use the fact that $N - M$ is small.

For indices $i \leq M$, the expected probability of deleting i is calculated correctly since $[i-d, i]$ is always wholly enclosed in the interval $[1, M]$.

For indices $i > M$, the issue occurs that the interval extends beyond M. Let us "assume" that our formula is still holding correct, and then subtract the extra probability added for the region extending beyond M.

Note that the excess region of $[i-d, i]$ is simply $\min(d+1, i-M)$. We know that $i - M \leq N - M \leq 20$. Hence, when $d \geq N - M$, the subtracted region will always be $i - M$.

For small $d < N - M$, we can store the number of ways of having an array with exactly d deletions in a dynamic programming. $dp_{i,j}$ = the number of arrays of length i with j deletions. Transitions are fairly trivial. We will only keep $j \leq N - M$.

Properly maintaining the values, we get a solution that works in $O(N \cdot \max(N - M, 1))$.

We now try to optimize to $O((N-M)^2)$. For this, our core goal is to skip the calculation of $dp_{i,j}$ for all $i \leq M$ and jump directly to $dp_{M,j}$ after which we can solve in quadriatic.

For this, we prove a powerful bijection.

Let $f(x_1, x_2, ..., x_k)$ $(1 \le x_1 < x_2 < ... < x_k)$ be the number of arrays with exactly the elements initially at positions x_i being deleted (and $M \ge N$ here).
We compute $f(x_1, x_2, ..., x_k) = (M-1)^{x_1-1} \cdot 1 \cdot (M-2)^{x_2-x_1-1} \cdot 2 \cdot (M-3)^{x_3-x_2-1} \cdot 3....$

Now, consider $g(x_1, x_2, ..., x_k)$ be the number of arrays with $1 \le A_i \le M$, $A_i \ne (k+1)$, each of $1...k$ appears at least once; and $x_1, x_2, ..., x_k$ being the **set of last positions** of $1, 2, ...k$ (in some order, not necessarily the same order)
We find $g(x_1, x_2, ..., x_k) = (M-1)^{x_1-1} \cdot K \cdot (M-2)^{x_2-x_1-1} \cdot (K-1) \cdot (M-3)^{x_3-x_2-1} \cdot (K-2).....$
Rearranging, $f(x_1, x_2, ..., x_k) = g(x_1, x_2, ..., x_k)$

Thus, we have a bijection. The number of arrays of length N and elements bounded by M $(M \ge N)$ and exactly d deletions is the number of arrays, also bounded by M and length N, such that $1...d$ appears at least once but $d+1$ does not.

We can now apply Principle of Inclusion Exclusion to compute $dp_{N-M,j}$ for each $j \le N-M$ in $O(N-M)$ complexity, achieveing $O((N-M)^2)$ complexity overall.

The expectation of elements $\le N-M$ can be handled as we handle $M \ge N$.

One final part is that we need to solve $M \ge N$ in $O(log(mod))$. We write the linear recurrence, and simplify it using some arithmetic/geometric sums to get an expression.

Problem F
Greedy Prices

Author: Shreyan Ray

Claim 1: We will use at most $log_2(10^9)+1$ items with $M_i, C_i > 0$

Proof 1: $M_i \cdot X + C \geq X$ for $M_i > 0$ and thus the cost at least doubles each step (after the first)

After fixing the set of the items we use, we can use Exchange argument to decide the order. Taking X and then Y costs $X.c + Y.m \cdot (X.c) + Y.c$, and taking Y and then X costs $Y.c + X.m \cdot (Y.c) + X.c$. Hence, by exchange argument, X before Y if and only if

$$Y.m \cdot X.c < X.m \cdot Y.c$$

$$\frac{X.c}{X.m} < \frac{Y.c}{Y.m}$$

. (Note that the relation is transitive for $M_i, C_i > 0$ so it's valid)

Now, assuming all items have $M_i, C_i > 0$, we can sort in the order of $\frac{C}{M}$ and then do dp_{ij} = minimum cost to buy j items from the prefix of length i. We use states $j \leq log_2(10^9)+1$, so this part is $N \cdot log(A)$.

For $C_i = 0$,all such items can be taken in the beginning without issue since they cost 0. Thus, just remove all such items and add the count to every answer.

For $M_i = 0$,all such items should be taken in the end. And after fixing the total cost used up in buying $M_i > 0$ items, we should take as many $M_i = 0$ items as possible. This means we should have sort by C_i and greedily take the largest prefix.

To answer queries,

- First iterate on $0 \leq j \leq log_2(10^9)+1$, the number of items with $M_i, C_i > 0$ that you will take.
- Binary search on the prefix sum of sorted array of $M_i = 0$ items to find how many extra items you can take.
- Maximise answer over all options
- Add number of $C_i = 0$ items to answer.

The complexity is $Nlog(A)log(N)$, and it can be improved to $Nlog(A)$ but it was not needed to pass.

Problem G
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.

Problem H
Majority Graph

Author: Satyam Kumar Roy

The problem is solvable with some techniques in $O(NlogN)$ or worse complexities. Here we present a non-standard $O(N)$ solution, which was the reason for keeping this problem.

Claim 1: We do not need to add edges (P, Q) where both $A_P \neq X, A_Q \neq X$ (X being majority of $A[P, Q]$).

Proof 1: Let R be first index in (P, Q) such that $A_R = X$ and S be last index such that $A_S = X$. Then, we will add the edges (R, Q) (R, S) and (P, S) so we can skip (P, Q).

Now, we will divide the problem into 2 parts, adding edges (P, Q) such that both $A_P = A_Q = X$ and **implicitly adding** edges such that $A_P = X \neq A_Q$.

1. First type of edge, i.e. $A_P = A_Q = X$

Turn the problem into an array B where $B_i = 1$ iff $A_i = X$, else $B_i = -1$. Let C be the prefix sums of B. Note that (P, Q) having majority X reduces to $C_Q > C_{P-1}$, or $C_Q \geq C_P$ since $A_P = X$.

Further, note that we only need to add edges such that $C_Q \leq C_P + 1$, other edges implicitly get added. Also, we only have to add to the first index Q such that $C_Q = C_P$ and the first index such that $C_Q = C_P + 1$. We only need to find the array C for the positions where $A_i = X$, which is simple to do. All the edges here can be generated in $O(N)$.

2. Second type of edge, i.e. $A_P = X \neq A_Q$

Note that the condition is clearly monotonic, i.e. $(P, Q - 1)$ will also be an added edge if $A_P = X \neq A_Q$.

Hence, we can reduce to: For each index i, find the maximum j such that there exists some P where (P, k) has majority X for each $i \leq k \leq j$.

Note that such a optimal P is fixed since it would just be the minimum C_P (using the same convention of C from above). We can then implicitly add edges of the form (P, k) by adding edges $(i, i + 1), ..., (j - 1, j)$. Here, each edge only needs to be added at most once, and we can maintain difference array to see which edges should be added. This part is also $O(N)$.

Finally, just perform a BFS on the graph formed to count the number of components. The time complexity is $O(N)$.

Problem I
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 $O((N + Q)logN)$.

Problem J
Max Mod

Author: Chongtian Ma

Note that, at any point, the array A will satisfy $A_i = x + iy$ for some x, y. So, the task at hand is to compute:

$$\max_{i=L}^{R} ((x + iy) \pmod M)$$

WLOG, we can assume $x = 0$, else, we can replace $[L, R]$ by $[L + xy^{-1}, R + xy^{-1}]$, where y^{-1} denotes the modular inverse of y wrt M. Hence, we need to maximize over $i \in [L, R]$:

$$(iy \pmod M) = iy - M \left\lfloor \frac{iy}{M} \right\rfloor$$

Note that, for a fixed $k = \lfloor \frac{iy}{M} \rfloor$, the above quantity is maximized for the largest such i, which is $f_k = \lceil \frac{(k+1)M}{y} \rceil - 1$, and the corresponding value would be:

$$\begin{aligned} f_k y - Mk &= y \left(\left\lceil \frac{(k+1)M}{y} \right\rceil - 1 \right) - Mk \\ &= (k+1)M + \big(-(k+1)M \pmod y \big) - y - Mk \\ &= M - y + \big(-(k+1)M \pmod y \big). \end{aligned}$$

Let $P = \lfloor \frac{Ly}{M} \rfloor$ and $Q = \lfloor \frac{Ry}{M} \rfloor$. Note that for $k = Q$, the best possible i is R. So, first account for that. Then, clearly for all $k \in [P, Q-1]$, $f_k \in [L, R]$. So, we need to maximize $((-M)k) \bmod y$ over all $k \in [P+1, Q]$.

Hence, we can recurse into smaller subproblems. This time complexity is good if $y \leq \frac{M}{2}$. Otherwise, note that we need to maximize $(R-i)y \mod M$ over $i \in [0, R-L]$. Or equivalently $(Ry + (M-y)i) \pmod M$. Hence, we can replace y by $M - y$. Thus, there will be $O(\log M)$ iterations.

Thus, the problem is solved in $O(Q \cdot log(M))$ or $O(Q \cdot log^2(M))$ depending on implementation.

Problem K
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 $O(N)$.

Problem L
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 $O(N\alpha(n))$.

Problem M
Ticket Revenue Maximization

Author: Jatin Yadav

Each round can be viewed as balanced bracketed sequence, where a (means they lost the round, and) means they won the round. Let us explain why this characterization is both necessary and sufficient.

First of all, the number of winners must be equal to losers of course. Suppose some suffix has positive balance [looking at (as +1,) as -1]. Then, there are more losers than winners here. But that is impossible as they cannot be paired up with that many stronger people.

Thus, every suffix has negative balance (or 0), and equivalently, every prefix has positive balance. Hence, it is necessary for it to be a balanced bracketed sequence, but is it sufficient?

We know in a balanced bracketed sequence, we can pair up the brackets into pairs of () such that the (always appears before the). This also gives us a valid pairing in terms of matches.

Now, let's extend this idea to multiple rounds. Let's call a person a x-loser if they lose in round x, and a person a x-winner if they win in round x.

The necessary and sufficient condition for the tournament to be valid is in every suffix, there are at least as many x-winners as x-losers.

We can do *dp* with our states being the number of x-losers for every x from 1 to 7, from back to front. Then, for the next person, we can decide which round we want to make him lose at, while ensuring the condition of every suffix having at least as many x-winners as x-losers is still true. The strongest person will always be the winner of all rounds.

Naively, this has $65 \cdot 33 \cdot 17 \cdot 9 \cdot 5 \cdot 3 \cdot 2$ states, which is around 10^7, however, you can note that a large amount of these are useless. This is because the suffix condition of more x-winners $\geq$ x-losers does not allow all states. The actual number of states is around $2 \cdot 10^5$.

Recursion with proper memoization, by unrolling the information into an array, can pass in ≤ 0.5s, while just putting the vectors into a map for memoization passes in around 2.5s.

Problem N
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 $O((N+M)log(N))$.