Problem A. Grid Problem

Notice that counting matrices from the statement is equivalent to counting matrices with row sums and column sums being divisible by 3. Sufficiency can be shown by obtaining 1×1 matrix with value 3 as a combination of matrices given in the input. After that you can obtain any remainders modulo 3 in $h-1 \times w-1$ submatrix of the grid and after that change values using 3 by keeping their remainders. Necessity is obvious. Now let's try to count matrices which satisfy this condition.

Let's consider polynomial

$$F(r_1, \ldots, r_h, c_1, \ldots, c_w) = \prod_{i=1}^{h} \prod_{j=1}^{w} \left(\sum_{t=0}^{k} (r_i c_j)^t \right)$$

Now notice this relation: $[r_1^{s_1}, \ldots, r_h^{s_h}, c_1^{q_1}, \ldots, c_w^{q_w}] F(r_1, \ldots, r_h, c_1, \ldots, c_w)$ is equal to the number of matrices
with row sums being equal to $s_1, \ldots, s_h$
and column sums being equal to $q_1, \ldots, q_w$. We want to count coefficients of terms with $s_1, \ldots, s_h, q_1, \ldots, q_w$ all being divisible by 3. To do that you can consider substitutions of non-trivial third root of 1. It exists since $\phi(10^9 + 9) = 10^9 + 8$ is divisible by 3. Let's denote the root as γ and $\Gamma = \{\gamma^0, \gamma^1, \gamma^2\}$. Then using the roots of unity trick which you can read about here https://codeforces.com/blog/entry/77551 you obtain formula for the answer:

$$\frac{1}{3^{h+w}} \sum_{\varepsilon_1, \ldots, \varepsilon_{h+w} \in \Gamma} F(\varepsilon_1, \ldots, \varepsilon_{h+w})$$

Let's expand the sum:

$$\sum_{\varepsilon_1, \ldots, \varepsilon_{h+w} \in \Gamma} F(\varepsilon_1, \ldots, \varepsilon_{h+w}) = \sum_{\varepsilon_1, \ldots, \varepsilon_h \in \Gamma} \sum_{\varepsilon_{h+1}, \ldots, \varepsilon_{h+w} \in \Gamma} \prod_{i=1}^{h} \prod_{j=1}^{w} \left(\sum_{t=0}^{k} (\varepsilon_i \varepsilon_{h+j})^t \right)$$

Notice that polynomial is symmetric, so we can only consider number of entries of each power of ω in variables representing columns and variables representing rows.

$$\begin{aligned} \sum_{\varepsilon_1, \ldots, \varepsilon_h \in \Gamma} \sum_{\varepsilon_{h+1}, \ldots, \varepsilon_{h+w} \in \Gamma} \prod_{i=1}^{h} \prod_{j=1}^{w} \left(\sum_{t=0}^{k} (\varepsilon_i \varepsilon_{h+j})^t \right) &= \sum_{\varepsilon_1, \ldots, \varepsilon_h \in \Gamma} \sum_{\varepsilon_{h+1}, \ldots, \varepsilon_{h+w} \in \Gamma} \prod_{j=1}^{w} \prod_{i=1}^{h} \left(\sum_{t=0}^{k} (\varepsilon_i \varepsilon_{h+j})^t \right) \\ &= \sum_{\varepsilon_1, \ldots, \varepsilon_h \in \Gamma} \left(\sum_{f=0}^{2} \prod_{i=1}^{h} \left(\sum_{t=0}^{k} (\varepsilon_i \gamma^f)^t \right) \right)^w \end{aligned}$$

Now we only need to notice that for $\varepsilon_1, \ldots, \varepsilon_h$ we only need to know occurences of each power, so the new formula becomes:

$$\begin{aligned} \sum_{\varepsilon_1, \ldots, \varepsilon_h \in \Gamma} \left(\sum_{f=0}^{2} \prod_{i=1}^{h} \left(\sum_{t=0}^{k} (\varepsilon_i \gamma^f)^t \right) \right)^w &= \sum_{0 \le c_0, c_1, c_2 \le h, c_0 + c_1 + c_2 = h} \binom{h}{c_0, c_1, c_2} \left(\sum_{f=0}^{2} \prod_{g=0}^{2} \left(\sum_{t=0}^{k} (\gamma^g \gamma^f)^t \right)^{c_g} \right)^w \\ &= \sum_{0 \le c_0, c_1, c_2 \le h, c_0 + c_1 + c_2 = h} \binom{h}{c_0, c_1, c_2} \left(\sum_{f=0}^{2} \prod_{g=0}^{2} \left(\sum_{t=0}^{k} \gamma^{(g+f)t} \right)^{c_g} \right)^w \end{aligned}$$

Since sum over k is geometric progression this sum can be efficiently computed in $O(h^2 \cdot \log k)$. Notice that γ^{g+f} might be equal to 1.

Problem B. Domain Compression

Note that any edge in the remaining graph is equivalent to some path in the initial graph from which all middle vertexes were removed. So let's calculate array p_i — number of paths of length i. We will count the length in number of vertexes between the two for which we consider the path. So adjacent vertices will be counted as a path with length 0. Now we are asked to calculate $f(k) = \sum_{i=0}^{n} p_i * \binom{n-i-2}{k-i} = \sum_{i=0}^{n} p_i * \frac{(n-i-2)!}{(k-i)!(n-k-2)!}$. This can be found using one convolution since all terms are separated into ones which depend only on k, ones which depend only on i and ones which depend only on $k-i$.

Problem C. Staple Stable

Notice that you only need to know maximal distance between some adjacent vertical cuts and maximal value between some adjacent horizontal cuts. Minimal of these values can't exceed $\sqrt{s}$ so by iterating over it we can find the answer in $O(\sqrt{s})$.

Problem D. Sequence Is Not Subsequence

Let's say that $ans(s)$ is answer to the problem for string s. We will assume s is of length at least 2 and we will now consider two cases:

- $s_0 = s_1$ in this case let's find first occurence of s_0 in $ans(s)$. Notice that suffix after this ocurrence is exactly $ans(s_1, \ldots, s_n)$ since it contains all needed substrings for suffix of s and does not contain the suffix itself.
- $s_0 \neq s_1$ in this case we also find first occurence of s_0 in $ans(s)$ and notice that suffix after that forms $ans(s_1, \ldots, s_n)$. However now it is necessary to make $s_1, \ldots, s_n$ a substring of $ans(s)$. To do so we can make a string $s_1 + s_0 + ans(s_1, \ldots, s_n)$. It can easily be seen that this string is optimal in terms of length.

This way answer can be found in $O(n)$.

Problem E. Coffee Shops

Notice that for any 2×2 square (if we write two arrays one under another) we always have at least 1 coffee shop which is not contained in the answer. This gives us $1.5n$ bound for the case of even n and $1.5n - 1$ bound for the case of odd n.

Turns out bound for odd n is achievable. After you put largest values in cells $a_1, a_3, \ldots, a_n$ all other cells are possible to fill in such a way that all remaining values contribute to the answer.

Bound for the even n is not possible to be achieved. You can notice that if we try to achieve bound of $1.5n$ then there are n cells which do not contribute to the answer and there is exactly 1 of such cells in each 2×2 square. If you try to fill in the rest of the values without losing any of them in the answer you will notice that this try will quickly terminate. It happens because all values which can be filled in will always have the same parity as parity of ends of the filled chain.

However value of $1.5n - 1$ good squares is achievable for even n. It can be done by putting largest values in cells $a_1, a_3, \ldots, a_{n-1}$ and b_1.

Problem F. Yet Another MST Problem

Let's try to create a modification of kruskal algorithm which works fast enough in this problem. Let's iterate over values $0 \ldots n$ and try to connect segments which we can currently connect with edge of current

value. We will denote currently considered value as v. Let's consider all segments which do not contain v. Then we can connect all such segments into one component using edges of value v. Now let's notice that for the new component we can say that it is equivalent to one segment $[a, b]$ that is an intersection of all connected segments. It is so because for any component of segments you only need to know if some value exists in all the segments.

To implement the procedure efficiently enough you can store all left and right borders of the segments in some sets. Solution works in $O((n + m) \log(n + m))$

Problem G. Cyclic Topsort

Let's first solve the problem without the special vertex (the one with index equal to 1 in the sequence). In this case, we can just greedily delete vertices with no ingoing edges and put them in this order in sequence v.

Going back to the original problem, we can notice that running the described above algorithm on the graph allows us to construct a prefix of the answer (not including the special vertex; we will just move it to the beginning of the v later). Now we only need to find maximal $f(u)$ where $f(u)$ defines how our answer changes when we make u the special vertex. Obviously, we do not need to consider already added vertices to be u since its suboptimal.

To quickly find $f(u)$, we will use the following trick: we will run our initial algorithm after adding u to the queue at the start. This way, we can find the answer to the problem in $O(n^2)$. To make it run faster, notice that each call to the initial algorithm works in $O(f(u))$ (if we implement it optimally enough). In this case, let's mark all visited vertices during the call of $f(u)$ (let's call this set as $s(u)$) as bad ones and do not start the calculation of f from them. We will now show that in case of shuffling the order of vertices and implementing the described procedure, the algorithm will work in $O(n \log n)$.

To show that, notice the following fact: for any pair of vertices u, w, the sets $s(u)$ and $s(w)$ are either disjoint or one contains the other. To prove that, let's consider any vertex o in their intersection (if such o does not exist, then $s(u)$ and $s(w)$ are disjoint). We know that o could not be added before marking some vertex as special, but it was removed when we marked u, w to be special. That means that there exists some path from u to o and that there exists some path from w to o. The only thing left to notice is that the only possibility is that there exists a path from u to w or that there exists a path from w to u. Otherwise, vertex o would not be possible to remove. Then its obvious that either $s(u) \subset s(w)$ or $s(w) \subset s(u)$.

That means that the structure of sets forms a tree, and every time you visit some vertex, you also mark its subtree as bad. Then you can count the expected number of times when some vertex u will be visited across all calls of f. That value behaves just like binary search over the path from u to the root of its tree. In the end, we get $O(m \log n)$ complexity.

Problem H. Misread Problem

Notice that problem actually asks us to minimize $\sum_{i,k} |a_{i,k} - b_i|$, where b is some correct stone distribution. Answer is equal to the value divided by 2. Now notice that function $f_i(x) = \sum_k |a_{i,k} - x|$ is convex as sum of convex functions and it can be represented as a set of slopes. If we restrict $f_i(x)$ only to the values of $x \in [0, m]$ constrain on b being valid distribution simplifies to b summing up to m. So we only need to find $\min_{b_1,\ldots,b_n} f_i(b_1) + \ldots + f_n(b_n)$. This is easily slope-trickable. Just insert all the slopes into a set and find the value at point m. Solution works in $O(nk \log nk)$.

Problem I. troS XEM

We will use the fact that for any segment of legnth 17 or more you can change it to any value in $[0, 3]$. This can be shown due to stress testing all possible segments. During the contest it could be noticed that length of the 17 is the minimal length for which array of only values 3 can construct all possible values. This case in some sense is the worst but since proving the fact formally is hard larger constants (up to 45) were allowed to pass if implemented properly.

Now to solve the problem we will count $dp_{i,k}$ — minimal number of operations needed, to make prefix of length i non-increasing with last value being k. Now notice that to recalculate $dp_{i,k}$ through values j such that $i - j \geq 17$ you only need to store some helper array since you can obtain any value with long segments. To recalculate $dp_{i,j}$ using small segments we need to learn procedure which checks whenever segment $(j, i]$ can obtain some value f using the operations.

To be able to check obtainability of values for small segments we will implement two dynamic programming procedures. For odd length segments we will store mask of obtainable values (stored value lies in $[0, 2^4)$) and for even length segments we will store mask of possible pairs of integers (for the sake of simplicity we store ordered masks so value lies in $[0, 2^{4\times4})$). This way we obtain solution which works in $O(n * K^2 * C)$, where $K = 17$ — our magic constant and C is some large constant needed to make transitions. To speed things up we can precalculate all transitions. After that solution becomes fast enough to pass.

Problem J. Yet Another Constructive Problem

Notice that if you have two subsequences a, b of length m with $lis(a) \leq lis(b)$ you can always construct a subsequence c of length m with any $lis(c) \in [lis(a), lis(b)]$. You can shown that fact by constructing some discrete process of change from a to b. For example you can remove one element from a and add one element to b. After each change lis changes by no more than 1 (it can decrease or increase or stay unchanged but difference is not larger than 1 by modulo). And due to discrete continuity you can find that c in $O(\log n)$ queries to finding lis, so c can be found in $O(n \log^2 n)$.

Because of that we only need to find subsequence of length m with maximal and smallest lis. Finding maximal is trivial since you can take lis of initial sequence and add/remove extra elements. Below we explain how to find subsequence with smallest possible lis.

First let's learn how to transform p into some normal form in which it is easy to construct minimal partitioning into decreasing subsequences. To do that we will try to find sequence of swaps of adjacent elements such that during each swap there is adjacent element with value in between the swapped elements. To do that we perform modification of the Robinson-Schnested-Knuth algorithm. The algorithm is explained in details here `https://codeforces.com/blog/entry/98167`. If you construct a sequence by writing elements from the bottom rows to the top rows of the tableau then the only thing left to notice that when adding new element to tableau you actually perform sequence of swaps which constructs normal form for the currently considered prefix of p. It is well known fact that Robinson-Schnested-Knuth algorithm constructs normal young tableau. Using that fact you can notice that top k rows form a sequence of maximal length which can be partitioned into k decreasing sequences. So you can pick smallest number of chains from the top until their length is at least m. Now you only need to come back to the original sequence. To do that we only need to show that those swaps we performed can be performed in reverse order with some changes to chains so they stay correct and have the longest length.

Let's say that we have elements with values $\alpha < \beta < \gamma$. Then allowed swaps are of the form:

- $\alpha, \gamma, \beta \to \gamma, \alpha, \beta$
- $\gamma, \alpha, \beta \to \alpha, \gamma, \beta$
- $\beta, \alpha, \gamma \to \beta, \gamma, \alpha$
- $\beta, \gamma, \alpha \to \beta, \alpha, \gamma$

For each of the swaps you can restore new chains by case analysis `https://www.sciencedirect.com/science/article/pii/0001870874900310`. Two do so efficiently you can store each chain in linked list which gives you solution in $O(n^2)$.

Problem K. Robot Construction

Notice that all possible heights after going through a segment always form a segment of values from 0 to maximum possible value. After encountering an obstacle of height h_i if all possible values form segment $[0, k]$ then k changes according to the following rules:

- If $k < h_i$ then nothing changes.
- If $h_i \le k < 2h_i$ then all possible values now form segment $[0, h_i]$.
- If $2h_i \le k$ then all possible values now form segment $[0, k - h_i]$.

Then let's solve the problem in offline. We will keep all values for some fixed r in segment tree and try to update them while going to $r + 1$. To update values you only need to notice that they are non-increasing. So you need to do nothing with prefix, make middle values equal to h_i and substract h_i from suffix. Finding boundaries of the changes can be done with descent on segment tree. So the problem is solveable in $O((n + q) \log n)$.