Problem A. Two Spanning Trees

For two sets of edges A and B we denote their difference by $A \setminus B$ and symmetric difference by $A \oplus B$. We identify corresponding edges in the graphs.

1. Find any spanning tree T of the first graph. T forms a spanning tree in the second graph; otherwise, it is an answer.

2. Reformulate the problem as follows: we are required to find a cycle C in the second graph, such that C forms a forest in the first graph (necessity is evident; for sufficiency, one needs to extend C to the spanning tree of the first graph).

3. Let e be an edge that does not belong to T. Denote $s_1(e)$ to be such a subset of T that $\{e\} \cup s_1(e)$ is a cycle in the first graph (the so called *fundamental cycle* of e). Define $s_2(e)$ analogously for the second graph. Note that $s_1(e) \subseteq s_2(e)$; otherwise, $\{e\} \cup s_2(e)$ is the answer.

4. Let E be a set of edges that do not belong to T. Define $\mathrm{Cyc}_1(E) := E \cup \bigoplus\limits_{e \in E} s_1(e)$. It is a cycle or a union of cycles. Define $\mathrm{Cyc}_2(E)$ analogously. Reformulate the problem as follows: we are required to find such a set E of the edges that do not belong to T, that $\mathrm{Cyc}_2(E)$ forms a forest in the first graph.

5. Let E be an answer. Let's write $e \sim e'$ in case $s_1(e) \cap s_1(e') \neq \varnothing$; let's write $e \approx e'$, if there exist $e_1, \ldots, e_N \in E$ such that $e \sim e_1 \sim \cdots \sim e_N \sim e'$. Partition $E = E_1 \sqcup \ldots \sqcup E_k$ according to the equivalence $\approx$. E is an answer; therefore, $\mathrm{Cyc}_2(E)$ doesn't contain cycles in the first graph; in particular, $\mathrm{Cyc}_1(E_1) \nsubseteq \mathrm{Cyc}_2(E)$. Now express

$$\mathrm{Cyc}_2(E) = \mathrm{Cyc}_1(E_1) \oplus \left(\bigoplus_{e \in E_1} s_2(e) \setminus s_1(e) \right) \oplus \left(\bigoplus_{e \in E \setminus E_1} s_2(e) \right).$$

   So there exists either an edge $e \in E_1$, such that $(s_2(e) \setminus s_1(e)) \cap \mathrm{Cyc}_1(E) \neq \varnothing$, or an edge $e \in E \setminus E_1$, such that $s_2(e) \cap \mathrm{Cyc}_1(E) \neq \varnothing$. In both cases, $(s_2(e) \setminus s_1(e)) \cap \mathrm{Cyc}_1(E) \neq \varnothing$.

6. Let's build a graph G with the vertex set being those edges of the two given graphs that don't belong to T. Draw an undirected edge in G between e and e' iff $e \sim e'$. Draw a directed edge $e \to e'$, iff $(s_2(e) \setminus s_1(e)) \cap s_1(e') \neq \varnothing$. Let E be an answer — a subset of G's vertices. The partition of E according to the equivalence $\approx$ is just the partition of the induced graph on E into connected components. By the previous clause, each connected component has at least one incoming directed edge.

   Therefore, the graph induced on E contains a cycle (a walk that can pass an undirected edge two ways, a directed edge one way, and passes through at least one directed edge).

7. Let's find the shortest cycle C in G. Next, we prove that C is an answer (that is, $\mathrm{Cyc}_2(C)$ forms a forest in the first graph).

8. Suppose that there are two edges of the form $e_1 - e_2$ and $e_1 \to e_2$ in G. Then simple casework shows that $\{e_1\} \cup \{e_2\}$ is an answer (we just need to check that neither of $\mathrm{Cyc}_1(\{e_1\}), \mathrm{Cyc}_1(\{e_2\}), \mathrm{Cyc}_1(\{e_1, e_2\})$ could be a subset of $\mathrm{Cyc}_2(\{e_1, e_2\})$).

9. Suppose that there are two edges of the form $e_1 \to e_2$ and $e_2 \to e_1$ in G. Analogously, $\{e_1\} \cup \{e_2\}$ is an answer.

   Now we can safely assume that there are no cycles of length 2 in G. In other words, for every e_1 and e_2, exactly one of the following holds: there is a directed edge $e_1 \to e_2$, a directed edge $e_2 \to e_1$, an undirected edge $e_1 - e_2$, or there are no edges between e_1 and e_2 at all.

10. The shortest cycle C in G contains no chords (no edges other than the ones between adjacent vertices — otherwise, it is easy to show that C is not the shortest one). Partition $C = C_1 \sqcup \cdots \sqcup C_\ell$ into connected (via undirected edges) components.

    Let's show that $\mathrm{Cyc}_1(C_1)$ is a cycle in the first graph (that is, it is connected). Since there are no chords in C, for all $e_1, e_2, e_3 \in C_1$ the intersection $s_1(e_1) \cap s_1(e_2) \cap s_1(e_3)$ is empty. Now, for the sake of contradiction, suppose the contrary: $\mathrm{Cyc}_1(C_1) = \mathrm{Cyc}_1(C_1') \sqcup \mathrm{Cyc}_1(C_1'')$ for some nonempty C_1', C_1''. Then there are two edges $e' \in C_1'$ and $e'' \in C_1''$, that are adjacent (linked by an undirected edge). This yields a contradiction: $s_1(e') \cap s_1(e'') \subseteq \mathrm{Cyc}_1(C_1')$, but $s_1(e') \cap s_1(e'') \nsubseteq \mathrm{Cyc}_1(C_1)$.

    For a fixed C_j, define e_j and f_j so that $e_j \in C$, $f_j \in C_j$, and there is a directed edge $e_j \to f_j$ (these conditions define e_j and f_j uniquely).

    Now we finally show that $\mathrm{Cyc}_2(C)$ doesn't contain cycles in the first graph. It suffices to show that $\forall C' \subset C : \mathrm{Cyc}_1(C') \nsubseteq \mathrm{Cyc}_2(C)$. Note that $\mathrm{Cyc}_1(C') \subseteq C \cup \bigcup\limits_{e \in C} s_1(e)$, and the last could be partitioned into biconnected components $C_j \cup \bigcup\limits_{e \in C_j} s_1(e)$. Therefore, such C', if exists, could only be contained in one component C_j. However,

    $$\mathrm{Cyc}_2(C) \cap \left(C_j \cup \bigcup_{e \in C_j} s_1(e) \right) = \mathrm{Cyc}_1(C_j) \setminus (s_2(e_j) \setminus s_1(e_j)),$$

    $\mathrm{Cyc}_1(C_j)$ is just a cycle, and $(s_2(e_j) \setminus s_1(e_j))$ non-trivially intersects with it. Therefore, the right hand side doesn't contain any cycles, which finishes the proof.

Since the problem doesn't require finding the answer, the answer is positive iff any directed edge of G lies on a cycle — that is, iff its endpoints are in the same strongly connected component. Therefore, to solve the problem, one needs to build graph G and condense it. It can be done in $\mathcal{O}(m^2)$, or (with slight modification of G) in $\mathcal{O}(nm)$.

Problem B. Christmas Tree

Let's root the tree in some vertex and solve the problem with subtree dp.

Define $U[v][i]$ as the minimal cost to direct all edges in a subtree of vertex v given that the edge between v and its parent is directed upwards and there are i vertices reachable from v from the outside of its subtree.

Define $D[v][i]$ as the minimal cost to direct all edges in a subtree of vertex v given that the edge between v and its parent is directed downwards and there are i vertices reachable from v.

(By the cost here we mean the contribution to the answer from the nodes in the subtree of v).

Suppose we are now in vertex v, let's fix some k and suppose we are given that there are k vertices reachable from v in total.

Having U and D for the children of vertex v, we can run a knapsack-like dp to find $h[v][l]$ — the minimal cost to direct all edges in a subtree of v so that there are l vertices reachable from v from its subtree, given that there are k vertices reachable from v in total.

Then $D[v][i]$ is $h[v][k]$ for $k = i$.

$U[v][i]$ is minimum of $h[v][k-i]$ over all $k > i$.

For a fixed k a total time to compute $h[v][l]$ over all vertices is $\leq Cn^2$, so the overall complexity is $O(n^3)$.

Problem C. Roman Numerals

Let's create two arrays, the i-th element of those stores value and priority of the i-th digit (sort the string representations of the digits, and use binary search, or just use `std::map` for that). After that, we can forget about the alphabet.

First solution

Note that the answer for the query (l, r) is $\sum\limits_{i=l}^{r} s_i v_i$, where v_i is the value of the i-th digit, and s_i is either $+1$ or -1. To calculate s_i, consider the stack of maximum records to the right of the i-th digit (formed by the links to the first element with strictly greater priority to the right). It could be proved by induction that s_i equals $+1$ iff the stack is of even size (for elements with the maximum priority, the stack is considered to be empty).

Let's answer the queries offline, iterating over them in the increasing order of the right boundary. Note that for a fixed right boundary r, all s_i are the same, regardless of the left boundary. Once r is incremented, all signs in the interval (p, r) are multiplied by -1, where p is the first element to the left of r with priority at least p_r.

Therefore, one could store the contributions of the digits in a lazy segment tree, that supports subsegment sum and subsegment multiplication. The complexity is $\mathcal{O}((n + q) \log n + (n + m) \log m)$.

Second solution

Consider the tree of maximums. Each vertex in it is responsible for some semi-interval $[l, r)$. The root of the tree is $[0, n)$. For a vertex $[l, r)$ with $l < r$, let $i \in [l, r)$ be the leftmost element with maximum priority. Then the vertex $[l, r)$ has the left child $[l, i)$ and the right child $[i + 1, r)$.

For a query $[l, r + 1)$, let i be the leftmost digit with maximum priority. Split the query into two queries of the form $[l, i)$ and $[i + 1, r + 1)$. These two new queries correspond to either prefix or suffix of some vertex in the maximum tree. To handle the queries that are prefixes or suffixes of some vertices, we traverse the tree and answer the queries that correspond to the prefix or suffix of the current segment upon leaving the vertex (so it's also offline).

Let's calculate the answers on prefixes of all vertices in the maximum tree. Consider a vertex that is responsible for some semi-interval $[l, r)$. Traverse both of its children $[l, i)$ and $[i + 1, r)$, and assume that the answers to $[l, j)$ and $[i + 1, j)$ are stored in the j-th position of some data structure. Then to convert these answers to the answers for $[l, j)$ prefixes, one needs to add $-\text{value}(d_l \dots d_{i-1})$ (it is stored in the $(i - 1)$-th position of the data structure) to the answers in the right half.

The answers for suffixes are calculated analogously, but one needs to be more careful with signs. For example, the author's solution passes into recursion the sign with which the answers should be stored.

To handle these operations, one needs a segment tree with range addition and queries of the value in a point. One also needs a sparse table or a maximum segment tree to calculate the tree of maximums in $\mathcal{O}(n \log n)$. The total complexity is also $\mathcal{O}((n + q) \log n + (n + m) \log m)$.

Problem D. Disjoint Set Splitting

Note that some prefix of the answers is 1, and the remaining suffix is 0. Assume that all the answers are 1. Then some queries would be decoded incorrectly, but it would never happen before the first zero. Therefore, the queries could be answered offline, assuming all the answers are zero, and then calculating the actual answers.

To handle the offline problem, find for each edge the first time it is removed (using either hashmap, map, or sorting and binary search; the fastest jury solution uses binary search and sorting, though it might be possible to make hashmap even faster), and then add the edges in the reversed order of their removal times. To find the first time of the graph being connected, one might use disjoint set union structure. The complexity is $\mathcal{O}((n + m + q)(\log m + \alpha(n)))$ or $\mathcal{O}((n + m + q)\alpha(n))$ depending on the implementation.

Problem E. Maximum Segment Sum

Let's store the current maximal sum on suffix *cur*. Then we either do $cur \to cur + 1$ or $cur \to \max(cur - 1, 0)$. So we are walking on $\mathbb{Z}$ numbered as $\dots, 3, 2, 1, 0, 0, 1, 2, 3, \dots$. Then we can find the number of ways for *cur* to not exceed k in $O(\frac{n}{k})$ by the reflection principle (by precomputing some sums on binomial coefficients firstly). So we can solve the problem in $O\left(\sum\limits_{k=1}^{n} \frac{n}{k}\right) = O(n\log(n))$.

Problem F. This Time I Will Be Lucky

Let's calculate the answer $f(a,b)$, then we have a formula $f(a,b) = max(0, (a-b+a\cdot f(a-1,b)+b\cdot f(a,b-1))/(a+b))$, then we can calculate $f(a,b)$ in $O(ab)$. How to do better. Let's only consider states (i,j) such that $|\frac{bi-aj}{a+b}| <= K$. (Close to the diagonal). Initialize other values by zero. Then we can solve in $O(max(a,b)\cdot K)$. Then we will have the mistake at most $(a+b)\cdot$(probability we are going out of this neighbourhood diagonal at some moment). By considering the random walking ξ which makes $+\frac{a}{a+b}$ with $p = \frac{b}{a+b}$ and $-\frac{b}{a+b}$ with $p = \frac{a}{a+b}$, showing that $\xi_{a+b} = 0$ with probability at least $\frac{1}{a+b+1}$ and using the Azuma inequality it can be shown that the mistake is at most $4(a+b+1)^2 e^{\frac{-2K^2}{max(a,b)}}$. So $K = 2000$ will be enough for this problem. (Even $K = 700$ is enough by the way).

Problem G. Far Away

Let's build DSU on our graph, and read the queries. If x_i and y_i are in the same component of size ≤ 20000 then answer is NO. Then let's 300 times select the vertex u randomly and run bfs from it, and for all queries with $dist(u,x_i) + dist(v,x_i) \leq 20000$ update answer as NO. For all other queries respond YES. If the answer for a query is NO, then it is either the first case, or there are at least 10000 vertices u that makes our answer for this query NO. So the probability we fail for a query is at most $(\frac{9}{10})^{300} \approx 1.9\cdot 10^{-14}$. So the probability we solve the problem wrong is at most $5.7\cdot 10^{-9}$.

Problem H. Absolutely Flat

Let b be an array of nonnegative integers. Fix a family of q segments, and define $f(b)$ to be the sum of oscillations of b on those segments.

For an array b and a nonnegative integer x, define another array b^x, such that $b_i^x = \begin{cases} 0, & b_i \leq x \\ 1, & b_i > x \end{cases}$. It is easy to see that $f(b) = \sum\limits_{x=0}^{\infty} f(b^x)$.

Now suppose that a is an array of nonnegative integers, where some elements are missing (we denote them below by ?). Define $g(a)$ to be the minimum of f over all arrays of nonnegative integers b that match a. It is easy to see that $g(a) \geq \sum\limits_{x=0}^{\infty} g(a^x)$, where a^x consists of 0, 1, and ?. In fact, there holds an equality (that is, the 01-principle works).

Proof

For two arrays c and d, let's write $c \leq d$ and $\min(c,d)$ in a componentwise meaning.

Let b_x be an optimal array that matches a^x (that is, b_x consists of zeroes and ones, matches a^x, and $f(b_x)$ is minimized). Define b_{x+1} analogously.

For any two arrays c and d of zeroes and ones, there holds an inequality $f(c) + f(d) \geq f(\min(c,d)) + f(\max(c,d))$ (which can be easily verified by considering $q = 1$). Therefore, $f(b_x) + f(b_{x+1}) \geq f(\min(b_x, b_{x+1})) + f(\max(b_x, b_{x+1}))$. On the other hand, b_x and b_{x+1} were the optimal arrays, and $\min(b_x, b_{x+1})$ also matches a^x. Hence $f(b_x) = f(\min(b_x, b_{x+1}))$ and $f(b_{x+1}) = f(\max(b_x, b_{x+1}))$. Hence one can replace b_{x+1} with $\min(b_x, b_{x+1})$. By induction, it could be assumed that $b_0 \geq b_1 \geq b_2 \geq \dots$, which is what we wanted.

Proof ends

The solution in $\mathcal{O}(n(q+n)\log n)$ goes as follows: by coordinate compression, one can assume that all the elements are from 0 to $n-1$. Now we need to deal with $\mathcal{O}(n)$ arrays, consisting of zeroes and ones. $f(b)$ is just the number of segments (from the given family of size q), such that they contain both 0 and 1.

If any segment already contains both zeroes and ones, then it is surely included in the cost, and we can dispose of it.

Note that it is never beneficial to fill some segment of ? with zeroes, provided both left and right of the segment are guarded with ones. For that reason, we can assume that the array is of the form ???000???111???000???111???, and any segment from the given family contains either only zeroes and questions, or only ones and questions.

We need to cut these segments of questions between zeroes and ones somewhere, and the cost of a cut placement is the number of segments that contain strictly inside at least one such cut (no segment could contain more than two cuts inside though).

The problem now could be solved via dynamic programming with a minimum segment tree, since the cost of placing cuts depends only on the pairs of two consecutive cuts.

To optimize the complexity, one needs to use the observation from the proof: for arbitrary x, it is safe to assume that $b_{x+1} \le b_x \le b_{x-1}$. Let's write a recursive function `solve(l, r, cuts)`, which returns the minimum cost of cuts placement, provided they are placed according to the boundaries, passed in the `cuts` array. The function should be called with parameters `l = 0`, `r = n`, `cuts` are the segments of question marks in the array a. The function works as follows: first of all, it solves the problem for $x = \frac{l+r}{2}$, and restores the answer. Then it splits all the requirements in the `cuts` array, and passes them to `solve(l, x-1)` and `solve(x+1, r)`. Segments of size ≤ 1 in `cuts` should be erased, since their contribution can be deduced independently of the dynamic programming.

It is also possible to implement the solution in $\mathcal{O}((n+q)\log n\alpha(n))$, since the dynamic programming solution requires only queries of adding 1 on suffix and querying global minimum, which could be done with DSU.

Problem I. Two Permutations

Define $p_{i,j}$ as the number of $k \le i$ such that $p_k \le j$ and $q_{i,j}$ analogously.

We can obtain q from p if and only if for every $1 \le i \le n$, $1 \le j \le n$, $p_{i,j} \ge q_{i,j}$.

It is necessary because every correct operation can only decrease the values of $p_{i,j}$.

To prove that it is sufficient we give an algorithm how to obtain q from p.

Suppose $p \ne q$ and x is the minimal number such that $p^{-1}(x) \ne q^{-1}(x)$.

Notice that $p^{-1}(x) < q^{-1}(x)$ since $p_{q^{-1}(x),x} \ge q_{q^{-1}(x),x}$.

Let's say p_k is the minimal number such that $p^{-1}(x) < k \le q^{-1}(x)$ and $p_k > x$. (such p_k always exist because $p_{q^{-1}(x)} > x$).

Notice that for $p^{-1}(x) \le l < k$ and $x \le y < p_k$, $p_{l,y} > q_{l,y}$. Indeed, let's assume the opposite, $p_{l,y} = q_{l,y}$. Then $q_{q^{-1}(x),y} > p_{q^{-1}(x),y}$, which is a contradiction.

So we can swap x with p_k.

Then repeat this process until $p^{-1}(x) = q^{-1}(x)$.

Then we move on to a larger value of x.

For a given x we can iterate over all values from $x+1$ to $p_{q^{-1}(x)}$ and every time when the current value v is between x and $p_{q^{-1}(x)}$ in the permutation p, we can swap x with v.

The overall complexity of this algorithm is $O(n^2)$.

Problem J. One Permutation

The sequence a_k is concave ($2a_k \ge a_{k-1} + a_{k+1}$).

The proof goes as follows:

a_k is the LCS of p and $1, 2, ..., n, 1, 2, ..., n, ..., 1, 2, ...n$ (k times).

This LCS can be found as the longest path in a graph on a grid $n \times kn$ with edges of weights 0 and 1 as in classical DP for the LCS problem.

We can draw two such grids $n \times (k-1)n$ and $n \times (k+1)n$ so that they will have a common center, then consider the optimal paths corresponding to a_{k-1} and a_{k+1}, then they will intersect and we can flip them to obtain some two paths in two $n \times kn$ grids with the sum of lengths $a_{k-1} + a_{k+1}$, so $2a_k \geq a_{k-1} + a_{k+1}$.

For each $0 \leq \lambda \leq n$ and $1 \leq k \leq n$, define $g(\lambda, k)$ as the maximum over all partitions of p into k subsegments (cost of the partition $-$ $k\lambda$).

Define $f(\lambda)$ as the maximum of $g(\lambda, k)$.

Define $k(\lambda)$ as the maximal $1 \leq k \leq n$ such that $g(\lambda, k) = f(\lambda)$.

Then $f(\lambda)$ and $k(\lambda)$ for a given λ can be found in $O(nlog(n))$ by dynamic programming with a Fenwick tree. $dp[i]$ is a maximal pair over all k over all partitions of the prefix $p_1, ..., p_i$ into k subsegments such that the lis of the last subsegment ends in p_i (cost of partition $-$ k λ, k).

Define $\lambda(k)$ as the maximal $0 \leq \lambda \leq n$ such that $k(\lambda) \geq k$. Actually, $\lambda(k) = a_k - a_{k-1}$ so it's enough to find $k(\lambda)$ for $0 \leq \lambda \leq n$.

Notice that $\lambda(0) = n, \lambda(n) = 1$, $k(\lambda) \leq 1 + n/\lambda$ and $k(\lambda + 1) \leq k(\lambda)$.

Using these properties, all values of $k(\lambda)$ can be found by D&C in $O(\sqrt{n})$ queries to find it for one λ (we will visit only $O(\sqrt{2^{-h}n})$ on the h-th level of recursion from the bottom), so the overall complexity is $n\sqrt{n}log(n)$.

Problem K. Game on Board

In the end all numbers divisible by gcd and at most maximum will appear because gcd will appear by Euclidean algorithm (If x and y ($x > y$) appears at some moment, then $x - y$ appears at some moment), and so $max, max - gcd, max - 2gcd, ..., gcd$ will appear. So we know the number of moves till the end ($max/gcd - n$) and just want to check whether this number is divisible by 3 or not.