Problem Tutorial: "Two Trees"

Build the centroid decompositions of both trees and denote them as C_1 and C_2. Consider some pair of vertices (v, u). Let w_1 and w_2 be the LCAs of u and v in C_1 and C_2, respectively. Let

$$f(v, w_1, w_2) = d(v, w_1, T_1) + d(v, w_2, T_2).$$

Note that

$$(d(v, u, T_1) + d(v, u, T_2))^2 = (f(v, w_1, w_2) + f(u, w_1, w_2))^2.$$

To calculate the answer for a fixed pair (w_1, w_2), store the following data in each vertex:

- $cnt[v]$ — the count of v's
- $sum_f[v]$ — the sum of $f(v, w_1, w_2)$
- $sum_{f^2}[v]$ — the sum of $f(v, w_1, w_2)^2$

The answer computation can be reduced to some queries of two types: add a vertex — update the above data; get some vertice's contribution to the answer — add

$$cnt[v] \cdot f(v, w_1, w_2)^2 + 2 \cdot sum_f[v] \cdot f(v, w_1, w_2) + sum_{f^2}[v]$$

to the answer. As for the order in which to process the vertices: fix w_1 and iterate over only those v's that lie in w_1's subtree in C_1. The time complexity of this solution is $O(n \log^2 n)$. Here is a link to a correct code for better understanding: https://ideone.com/ys8hqj.

Problem Tutorial: "Tarzan Jumps"

Let ans_k be the answer for k. If we set H_1 and H_N to 0, then Tarzan will be able to reach the last tree in 1 jump. Thus, $ans_k \leq 2$ for any k. So, $ans_k = 0, 1$ or 2. Since $ans_k \geq ans_{k-1}$, then the array ans will look like this: $2, 2, \ldots, 2, 1, 1, \ldots, 1, 0, 0, \ldots, 0$. It follows that it is enough to find such minimal p_0 and p_1 that $ans_{p_0} = 0$ and $ans_{p_1} = 1$.

1) How to find p_0.

Consider a jump from tree x to tree y in which all the intermediate trees are lower than both of those trees (the other case is similar). There are two possibilities:

- $h_x \leq h_y$. Then y is the first tree to the right of x, which is not lower than x.
- $h_x > h_y$. Then x is the first tree to the left of y, which is not lower than y.

Hence, the number of pairs (x, y) such that the jump from tree x to tree y is possible is of magnitude $O(n)$. For each tree, we can find the closest tree to the right (and left) of it, which is not lower (higher) than it (using a stack). Using this information, we can compute A_i — the minimum number of jumps required to reach the i-th tree. Clearly, $p_0 = A_n$.

2) How to find p_1.

Let B_i be the minimum number of jumps to reach tree N starting from tree i (it can be computed in the same way as A_i). Let pos be the tree the height of which we change.

There are two cases:

1. The shortest path from 1 to N does not visit pos. The jump from i to j after one change is possible if initially one of the two conditions was held:
   (a) There was at most one tree between i and j with height at least $\min(h_i, h_j)$.

   (b) There was at most one tree between i and j with height at most $\max(h_i, h_j)$.

   Let's consider the first case (the second case is similar, you can just multiply all the numbers by (-1)).

   Let l_i be the second tree to the left of i that is not lower than tree i (if there is no such tree, then let $l_i = 1$). Let r_i be the second tree to the right of i that is not lower than tree i (if there is no such tree, then let $r_i = n$). Tarzan can jump from i to j after one change if and only if $l_j \le i$ and $j \le r_i$ (we need to change the highest tree between i and j for enabling this jump). Among such i, j we need to find those for which $A_i + 1 + B_j$ is minimal. It can be done with a standard scanline algorithm and a segment tree. Initially, in leaf i we store the value A_i. After that, we iterate from $j = 1$ to $j = n$. We update p_1 with $getmin(l[j], j) + B_j + 1$. And finally, for every i such that $r_i = j$, in leaf i we change the value to INF.

2. The shortest path from 1 to N visits pos. Let i and j be such that in the shortest path Tarzan jumps from i to pos, and then from pos to j.

   There are 4 cases:

   (a) All trees between i and pos are lower than i and pos, and all trees between pos and j are lower than pos and j. We can safely set $H_{pos} = INF$. It is possible to jump from i to pos if and only if i appeared in the stack from part 1 at the moment before we processed tree pos (the stack which was used to find the closest from the left tree that is not lower than the current tree). To find an optimal i, we can maintain all values A_x in a set, where x is a tree on the stack. This way, we can find i with minimal A_i, from which Tarzan can jump to pos. Similarly, find j with minimal B_j and update p_1 with $A_i + B_j + (pos \ne 1) + (pos \ne n)$.

   (b) All trees between i and pos are higher than i and pos, and all trees between pos and j are higher than pos and j. It is similar to case (a).

   (c) All trees between i and pos are lower than i and pos, and all trees between pos and j are higher than pos and j. For this to happen the following condition must hold:

   $$\max(H_{i+1}, H_{i+2}, .., H_{pos-1}) + 1 < \min(H_{pos+1}, .., H_{j-1}).$$

   Since

   $$\max(H_{i+1}, H_{i+2}, .., H_{pos-1}) < H_i,$$

   then the total number of possible values of max for all i's will be of magnitude $O(n)$. Let's iterate over all pairs (i, k) (such that $H_k = \max(H_{i+1}, H_{i+2}, .., H_{pos-1})$). It is optimal to choose pos as the first tree to the right of k that is not lower than H_k. Set $H_{pos} = H_k + 1$. Let pos_2 be the first tree to the right of pos that is not higher than $H_k + 1$. Since

   $$\max(H_{i+1}, H_{i+2}, .., H_{pos-1}) + 1 < min(H_{pos+1}, .., H_{j-1}),$$

   then $j \le pos_2$. j must satisfy the following conditions:

   $$pos < j \le pos_2, L_j \le pos,$$

   where L_j is the first tree to the left of j that is not higher than H_j. Among all such j's we need to find one with minimal B_j, and update p_1 with $A_i + B_j + 2$. This can be done offline with a segment tree.

   (d) All trees between i and pos are higher than i and pos, and all trees between pos and j are lower than pos and j. It is similar to case (c).

Problem Tutorial: "Inversions"

Let S_n be the set of all permutations of length n, π — an element of S_n, $I(\pi)$ — the set of inversions in π ($|I(\pi)| = inv(\pi)$), and

$$M_j(\pi) = \{((\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_j, \beta_j)) \mid (\alpha_i, \beta_i) \in I(\pi)\}, \quad |M(\pi)| = inv(\pi)^j.$$

Also, let

$$D_j(\pi) = \{\{(\alpha_1,\beta_1),(\alpha_2,\beta_2),\ldots,(\alpha_j,\beta_j)\} \mid (\alpha_i,\beta_i) \in I(\pi),\ \forall i < t:\ (\alpha_i,\beta_i) \neq (\alpha_t,\beta_t)\},$$

and

$$G_j(\pi) = \{d \mid d \in D_j(\pi),\ \{\alpha_1,\beta_1,\ldots,\alpha_j,\beta_j\} = \{1,2,\ldots,|\pi|\}\}$$

Then the answer can be interpreted as

$$|\{(\pi,m) \mid \pi \in S_n,\ m \in M_k(\pi)\}|.$$

Let

$$a_{i,j} = |\{(\pi,d) \mid \pi \in S_i,\ d \in D_j(\pi)\}|, \quad 1 \le i \le 2k, \quad 0 \le j \le k.$$

The following recurrence relation holds:

$$a_{i,j} = \sum_{q=0}^{j} a_{i-1,j-q} \cdot \left(\binom{0}{q} + \binom{1}{q} + \ldots + \binom{i-1}{q}\right) = \sum_{q=0}^{j} a_{i-1,j-q} \cdot \binom{i}{q+1}.$$

Think of it as choosing a position for i. Let

$$A_i(x) = \sum_j a_{i,j}x^j, \quad Q_i(x) = \sum_q \binom{i+1}{q+1} x^q,$$

then

$$A_i(x) = A_{i-1}(x) \cdot Q_{i-1}(x).$$

Let

$$b_{i,j} = |\{(\pi,d) \mid \pi \in S_i,\ d \in G_j(\pi)\}|, \quad 1 \le i \le 2k, \quad 0 \le j \le k.$$

It satisfies the following recurrence relation:

$$b_{i,j} = a_{i,j} - \sum_{t=1}^{i-1} b_{i-t,j} \binom{i}{t}^2 t!$$

Think of it as choosing the elements that are *not covered* by some $d \in D_j(\pi)$. Let

$$f(i,t) := \binom{i}{t}^2 t!, \quad b_{i,j} = a_{i,j} - \sum_{t=1}^{i-1} b_{i-t,j} f(i,t).$$

Rewrite the last formula:

$$b_{i,j} = \sum_v \sum_{t_1+\ldots+t_v \le i} (-1)^v a_{i-t_1-\ldots-t_v,j} f(i,t_1) f(i-t_1,t_2) \ldots f(i-t_1-\ldots-t_{v-1},t_v) =$$

$$= \sum_v \sum_{t_1+\ldots+t_v \le i} (-1)^v a_{i-t_1-\ldots-t_v,j} \cdot \frac{i!^2}{t_1!\ldots t_v!(i-t_1-\ldots-t_v)!^2} =$$

$$= \sum_v \sum_{t_1+\ldots+t_v \le i} i!^2 \cdot \frac{a_{i-t_1-\ldots-t_v,j}}{(i-t_1-\ldots-t_v)!^2} \cdot \frac{(-1)^v}{t_1!\ldots t_v!}.$$

Let

$$A_j^*(x) = \sum_i \frac{a_{i,j}}{i!^2} x^i,$$

$$B_j^*(x) = \sum_i \frac{b_{i,j}}{i!^2} x^i,$$

$$C(x) = \sum_i \left(\sum_v \sum_{t_1 + \ldots + t_v = i} \frac{(-1)^v}{t_1! \ldots t_v!} \right) x^i.$$

Then

$$B_j^*(x) = A_j^*(x) \cdot C(x).$$

So, we can find $b_{i,j}$'s using FFT in $O(k^2 \log(k))$ time. After that, computing the answer is straightforward.

If $n \geq 998\,244\,353$, the answer is zero. Otherwise, we will have to compute $n!$ one time. It can be done by storing some precalculated factorials in the source code.

Complexity: $O(k^2 \log(k))$

Problem Tutorial: "Hidden Rook"

TL;DR There are a lot of special cases in this problem. Apart from them, the main idea is the following. Ask about a rectangle in a corner whose sides are not equal and are approximately halves of the current big rectangle's dimensions. Based on the answer, go into one of the four subrectangles, but store some information about the previous rectangle, as it may be needed in the future. Don't forget to *think outside the box.*

Problem Tutorial: "Mountains"

Solution. Consider such an array $a[1..n, 1..m]$ (that represents a valid map). Let's calculate a standard DP:

$$dp[i,j] = a[i,j] + \max(dp[i+1,j], dp[i,j+1]),$$

which stores the largest sum over all paths from (i,j) to (n,m). We can observe two facts:

- $dp[1,1] \leq k$;
- $dp[i,j] \geq dp[i+1,j]$ and $dp[i,j] \geq dp[i+1,j]$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Let's view matrix dp as a pile of cubes: put $dp[i,j]$ cubes on top of the cell (i,j). (see fig. 1). Such an object is called a *plane partition*. We will reduce the initial problem to the problem of counting plane partitions in a box $n \times m \times k$.

$$\mathrm{a} = \begin{array}{|c|c|c|}\hline 1 & 0 & 2 \\ \hline 0 & 3 & 0 \\ \hline\end{array} \qquad \mathrm{dp} = \begin{array}{|c|c|c|}\hline 4 & 3 & 2 \\ \hline 3 & 3 & 0 \\ \hline\end{array}$$

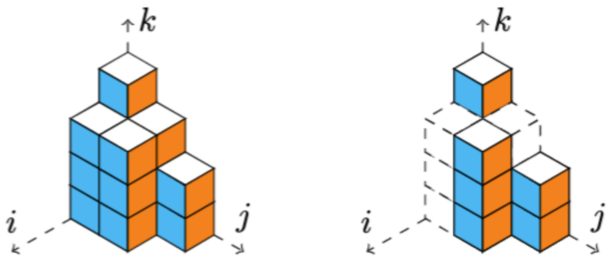


Figure 1. From left to right: array $a[i,j]$, dp on array a, dp as plane partition, 'corner flats' map to entries of a.

First, let's understand that the map $a \to dp$ is a bijection between $n \times m$ matrices with largest path $\leq k$ and plane partitions in a box $n \times m \times k$. To show that, we can provide a simple inverse map: suppose that you are given a plane partition $dp[i,j]$, then

$$a[i,j] = dp[i,j] - \max(dp[i+1,j], dp[i,j+1]).$$

To understand the meaning of entries $a[i,j]$ in terms of a plane partition, take a look at the rightmost picture of Fig. 1.

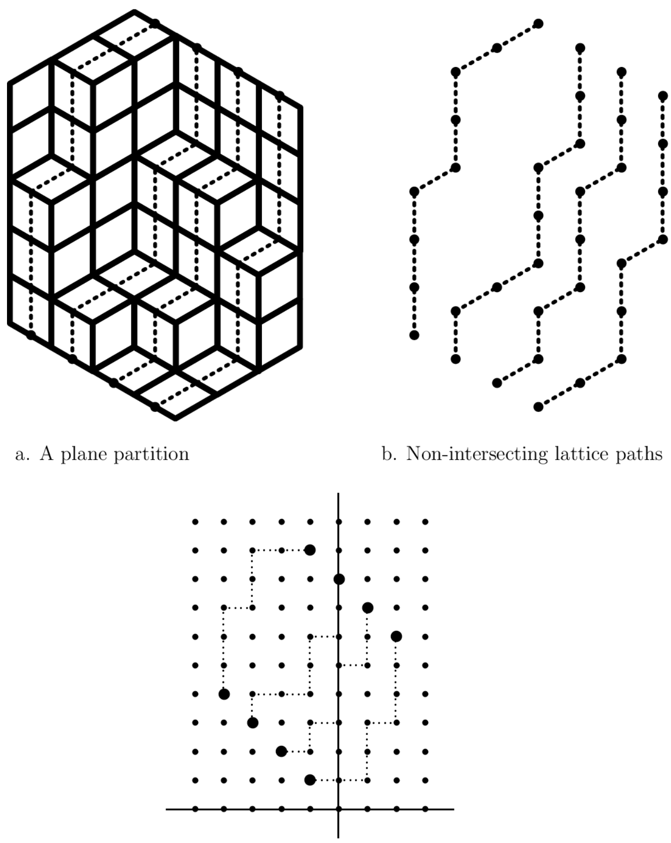
Now, to calculate the number of plane partitions in a box we can use several methods. One of the coolest is to view an arbitrary plane partition as a non-intersecting path system in a way represented in Fig. 2. Non-intersecting path systems in any planar directed acyclic graph can be enumerated by calculating the determinant of the 'path' matrix $f[i,j]$ = number of paths from $i \to j$ in the graph (by Lindstrom-Gessel-Viennot lemma). This can be done in $O(N^3)$ time with the Gaussian elimination algorithm.

Alternatively, entries $f_{i,j} = \binom{m+n}{n+j-i}$ are in fact binomial coefficients, so you can calculate the answer explicitly. This approach leads to a beautiful formula:

$$PP(n,m,k) = \prod_{i=1}^{n}\prod_{i=1}^{m}\prod_{i=1}^{k}\frac{i+j+k-1}{i+j+k-2},$$

which can be calculated in $O(N)$ time, but this was not required.

Complexity: $O(n^3)$



a. A plane partition

b. Non-intersecting lattice paths

Figure 2. Correspondence between plane partitions in a box and non-intersecting path systems.

Open problem. Calculate $PP(100,100,100,100)$, i.e. number of DPs $dp[i,j,k] \le 100$ with similar inequalities in all directions.

Problem Tutorial: "Kill All Termites"

We need to poison the minimum number of vertices so that any path between any two leaves contains a poisoned vertex. Run a DFS from a vertex of degree at least 2 and do the following: if v is a leaf then set $a[v] = 1$, otherwise set $a[v] = \sum_{u\text{ - son of }v} a[u]$, and if $a[v] > 1$ then poison this vertex and set $a[v] = 0$. This way we ensure that each path between leaves contains a poisoned vertex. On the other hand, if we don't poison some vertex v with $a[v] > 1$, then there certainly is a safe path for termites.

Problem Tutorial: "Maximal Subsequence"

This problem was taken from AITU Open Spring 2021.

Let L be the beauty of the given array, and

$$l_i = \text{ the length of the longest increasing subsequence ending with } a_i,$$

$$r_i = \text{ the length of the longest increasing subsequence starting with } a_i.$$

Note that it is reasonable to only delete those i's for which $l_i + r_i - 1 = L$. Let M be the set of all such i's. Let

$$V = \{S, T\} \cup \{i_{in}, i_{out} \mid i \in M\},$$
$$E = \{(S, i_{in}, 1) \mid l_i = 1\} \cup \{(i_{out}, T, 1) \mid l_i = L\} \cup \{(i_{out}, j_{in}, 1) \mid l_i + 1 = l_j,\ a_i < a_j\},$$
$$G = (V, E) \quad (G \text{ is a flow network}).$$

It is easy to see that our problem is equivalent to finding the minimum cut of G. Since the minimum cut is equal to the maximum flow, then our problem reduces to the following: find the maximum number of pairwise disjoint increasing subsequences of length L. It turned out that exactly this problem appeared in a recent contest. You can find a greedy algorithm that solves it here (with proof): `https://dmoj.ca/problem/coci21c1p5/editorial`.

Problem Tutorial: "Aidana and Pita"

We will use the "meet in the middle" approach. Let's split the given array into two halves of size $\frac{n}{2}$ and for each part consider all possible $3^{n/2} < 1.6 \cdot 10^6$ distributions of dishes into 3 groups. Let L_1 and L_2 be the lists of sums (x, y, z) in the left and right parts. Let (x_1, y_1, z_1) be from L_1 and (x_2, y_2, z_2) be from L_2. Without loss of generality, $x_1 + x_2 \geq y_1 + y_2 \geq z_1 + z_2$. Now we want to minimize $x_1 + x_2 - z_1 - z_2$.

Transform each triplet in L_1 as $(x_1, y_1, z_1) \to (x_1 - y_1, y_1 - z_1)$ and each triplet from L_2 as $(x_2, y_2, z_2) \to (y_2 - x_2, z_2 - y_2)$. Then it is not hard to see that for a pair of points $(a_1, b_1) \in L_1$ and $(a_2, b_2) \in L_2$, we want to minimize $a_1 + b_1 - a_2 - b_2$ having $a_1 \geq a_2$ and $b_1 \geq b_2$.

Merge the lists L_1 and L_2 into L and sort L in increasing order of the first coordinate. We will iterate over L and find the best fit for a fixed $(a_1, b_1) \in L_1$. To do so, we maintain a set S of pairs $(b, a + b)$ of points from L_2 with both coordinates increasing. So we iterate over pairs $(a, b) \in L$ and think:

- if (a, b) is from L_1, then take from S a point with maximal second coordinate (with max $a + b$),
- if (a, b) is from L_2, then remove all the points $(b', a' + b') \in S$ with $b < b'$ but $a + b > a' + b'$, and insert $(b, a + b)$ into S.

Then at each moment of time set S will contain the points of L_2 in weakly increasing order of both coordinates since we move in increasing order of a and remove non-optimal solutions when needed.

Time complexity: $O(3^{n/2} n)$

Problem Tutorial: "Box Packing"

Let's sort the pairs (a_i, b_i) lexicographically. Then the problem is reduced to finding the largest number of elements that can be split into k disjoint weakly increasing subsequences of b. To solve this, we will use the RSK algorithm.

First, recall the classical Longest Increasing Sequence search algorithm that stores $d[i]$ = the smallest value of the last element of an increasing subsequence of length i. The algorithm then does the following:

1. initially, vector d is empty;
2. insert the numbers one by one. To insert x, we find the smallest j such that $x < d[j] = y$, then x 'bumps' into y, i.e. we set $d[j] = x$. Otherwise, if x is not smaller than any number in the row, we append it to the end. As a result, vector d will be weakly increasing, so we can find a suitable j with binary search.

3. after all numbers are inserted, the size of d is the answer.

RSK is an extension of this algorithm. We now have not a single row d, but several rows $d_1, d_2, \ldots$. Initially, they are all empty. Similarly, we insert the numbers one by one as in (2), but instead of forgetting about the bumped value y, we try to insert it into the next row. This value will be either appended to its end or bumped yet into another number which is again inserted into the next row and so on.

As a result, we get a sequence of rows $d_1, \ldots, d_m$ such that the entries weakly increase in each row and strictly increase in each column (why?). Let $\lambda_i = size(d_i)$, then λ is called a partition of n (sum of lengths is equal to n), a.k.a a Young diagram if drawn as λ_i empty boxes in row i. Let λ' be a partition of the same diagram but by columns (for example, if $\lambda = (3, 2, 2)$ then $\lambda' = (3, 3, 1)$). It turns out that:

Theorem.[Greene 1974] Let λ be a diagram that we get after applying the RSK algorithm to a word $b_1, \ldots, b_n$. Then

- $\lambda_1 + \ldots + \lambda_k$ is equal to the largest number of entries in the union of k weakly increasing subsequences;
- $\lambda'_1 + \ldots + \lambda'_k$ is equal to the largest number of entries in the union of k strictly decreasing subsequences.

Therefore, we need to store only the first k rows and ignore the numbers going to the $k + 1$-st row.

Complexity: $O(nk \log(n))$

See the proof in *Stanley, Enumerative Combinatorics Vol. 2, Appendix A*. Unfortunately, it is non-trivial; finding a more direct argument would be interesting.

Proof sketch. Let w be a word and $RSK(w) = P$ of shape λ and let $I_k(w)$ be the answer for the initial problem. Show that the statement is true for a permutation (or word) w_0 that is formed by reading the tableau from bottom to top appending row by row (called reading a word). Clearly, $RSK(w_0) = P$. We show first that $I_k(w_0) = \lambda_1 + \ldots + \lambda_k$. Then we show that $RSK(w) = RSK(w_0)$ if and only if $w \sim w_0$, where $\sim$ is the so called Knuth equivalence relation on words. Finally, we prove that $w \sim w_0$ and I_k are preserved by the Knuth transformations.

Remark. The discussion in a Telegram PTZ chat yielded an interesting question: how to restore the answer? One approach could be the following. First, find the answer for w_0 (the i-th row of $RSK(w_0)$ is the i-th subsequence). Then find the sequence of Knuth moves from $w_0 \to w$ and change the answer according to them.

Problem Tutorial: "Two Permutations"

First of all, let's think of our permutations as a matrix with two rows and fill it in increasing order. For example, after placing the first 3 numbers our matrix will look like this:

1	2			3
3	1		2	

For this matrix, let x be the number of complete columns, y be the number of columns where only the upper cell is filled, and z be the number of columns where only the lower cell is filled. The fact that we fill the cells in increasing order of numbers implies that the last placed number is immediately the greatest in the column and we can add it to the sum. Also, notice that after we place all the numbers from 1 to some i, y is equal to z.

Let $dp[i][x][s]$ be the number of ways to place the numbers from 1 to i in the matrix so that there are x complete columns and the current sum is s, then

$$y = z = i - x$$

and we perform the transitions as follows:

- we can put both of the i's in the empty columns:

$$dp[i+1][x][s] += dp[i][x][s];$$

- we can create one complete column and there are $y+z$ ways to do that:

$$dp[i+1][x+1][s+i] += dp[i][x][s] \cdot 2(i-x);$$

- we can create two complete columns and there are x^2 ways to do that:

$$dp[i+1][x+2][s+2i] += dp[i][x][s] \cdot x^2.$$

The last step is to optimize the memory usage. Note that to calculate any $dp[i+1][x][s]$ we don't need to know $dp[i-1][x'][s']$. So at any given moment we just need to store two $n \times k$ matrices.

Complexity: $O(n^2 k)$

Problem Tutorial: "Fancy Arrays"

Let's first consider a suboptimal solution. We write down all divisors of m as $d_1 = 1, \ldots, d_k = m$ and create a matrix $D[i,j] = [\gcd(d_i, d_j) > 1]$. Then the answer is equal to the sum of entries of vector $(0, \ldots, 0, 1) \cdot D^n$. It can be computed with fast matrix exponentiation.

We now need to reduce the dimensions of that matrix. Let's think of what we really care about in transitions from one divisor to another:

- We can treat all primes with equal occurrences $\alpha_i = \alpha_j$ as indistinguishable. Let us group them together in $C_1, \ldots, C_g$, and for each group remember the corresponding occurrence α_i;
- For each prime we only care if it is present in adjacent divisors, so we may think $\alpha_i = 0$ or 1 and store the number of non-zero powers. So for each group C_i we can store the number of 1s in group C_i satisfying $0 \le b_i \le |C_i|$.

After these optimizations, we will have $N \le 255$ states, each of which can be described as $(b_1, \ldots, b_g)$ where b_i denotes the number of primes from group C_i with non-zero occurrences. Let i and j be the states described above. Note that for a fixed divisor d_1 of state i, the number of possible divisors d_2 of j is the same for all d_1 from state i. Hence, we set $D[i,j]$ to be the number of transitions between states i and j, which can be calculated combinatorially by "ALL - BAD" principle, and do matrix exponentiation of an $N \times N$ matrix D.

Finally, to answer the queries we can precalculate all powers of matrix D that are powers of 2 $(D^1, D^2, D^4, D^8, \ldots)$ and answer each query in $N^3 \log(n)$ time.

Complexity: $O(qN^3 \log(n))$

Problem Tutorial: "Restricted Arrays"

Let G be a graph on the elements of an array that satisfies the given conditions. For each condition (x, y), let's draw a directed edge $x \to y$ with weight -1 and an edge $y \to x$ with weight $+1$. Let's consider some vertex v_0. Without loss of generality, $a[v_0] = 0$ (if we have some valid array a for some M then we can shift all the elements by $-a[v_0]$ and get another valid array). Start a DFS in v_0 and calculate $a[v]$ for each v in the connected component of v_0. This way you will satisfy the rules corresponding to the direct edges from the DFS tree.

To satisfy a backward edge (v, u) we want $abs(a[v] - a[u] + w(v,u)) \bmod M == 0$ where $w(v,u)$ is the weight of edge (u, v).

Thus, any valid M must be a divisor of

$$N = \text{GCD}_{(v,u)\text{ - a backward edge of the DFS}}(abs(a[v] - a[u] + w(v,u)))$$

So, calculate N for each connected component, take the GCD of all Ns, store it in N_0. Then the answer would be all the divisors of N_0.

Time complexity: $O(n + q \cdot \log(n))$.