WWPIT 2025 Editorial

WWPPC

April 2025

1 Beat the Level

We can keep a counter that increments when it sees S, and resets to 0 when it sees A. If $S > 3$, then the code outputs DIE, otherwise it will output BEAT.

Alternate solution: Simply check if "SSSS" is a substring of the input string.

2 Stereo Madness

Expanding the equation, we observe a pattern: $\frac{a_2}{a_1} \cdot \frac{a_3}{a_2} \cdot \frac{a_4}{a_3} \cdot \frac{a_5}{a_4} \cdots \frac{a_n}{a_{n-1}}$.

We can cancel out the adjacent terms in the numerator and denominator. making the simplified result of the product $\frac{a_n}{a_1}$.

Assuming the array is unordered and we want to maximize this ratio, the best we can do is: $\frac{\max(a)}{\min(a)}$.

This can be implemented easily in code by using a loop (or built-in functions) to find the maximum and minimum elements in the array, and then dividing them. Make sure to print the float to enough decimal places.

3 Base After Base

Let's break down the expression. If we are evaluating it for A and B, consider each individual bit of A and B. Writing the truth tables (note that $a \ll 1 = a \cdot 2$):

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

A	B	$(A \mid B) \ll 1$
0	0	0
0	1	0
1	0	0
1	1	2

A	B	$A \oplus B + ((A \mid B) \ll 1)$
0	0	0
0	1	1
1	0	1
1	1	2

In fact, the given expression is equivalent to $A + B$, so we can simply print the indices of the two largest elements.

4 Keymaster

The naive solution is to iterate through each tax bracket for each query; however, this will take $O(nq)$ time, which is too slow given the constraints.

To compute the answer faster, we can create a prefix sum of the amount of tax we need to pay if earnings are above a certain amount. Then, for each query, we can binary search on this prefix sum to find how much we have to pay.

Alternatively, we can sort the queries and answer them in increasing order, which allows us to only pass through the array a single time.

5 Designer 2.3

Note that the smallest value in the grid must be $a_{0,0}$ and the largest value in the grid must be $a_{n-1,m-1}$. Then the second smallest value in the grid must be $a_{0,1}$ or $a_{1,0}$. From just these three values, we can calculate the common difference both row and column wise, so we are able to fill in the rest of the grid. We can try both possibilities and see which one works. To check quickly, convert the array into a multiset and extract values to fill the grid elements.

6 Cycles

Let's rearrange the formula $a_i \cdot a_k = a_j + a_k \rightarrow (a_i - 1) \cdot a_k = a_j$. Let's fix the center element a_j. Notice that both $a_i - 1$ and a_k are factors of a_j. This means we can iterate through the factors $x|a_j$ and count the number of $i < j, a_i = x+1$ times the number of $j < k, a_k = \frac{a_j}{x}$. We can keep track of both of these counts using an array to track the counts of the numbers before and after, and multiply them. There are at most $O\left(N^{\frac{1}{3}}\right)$ factors of N so the overall time complexity of this solution is $O\left(N^{\frac{4}{3}}\right)$, which is fast enough to pass. Note that you have to use a sieve to find all the factors of each number from 1 to N so that you can quickly iterate through factors.

7 Levels and Coins

Sort the levels by a_i. Note that each level will either be at max capacity or have just under x coins (or both).

For each $1 \leq r \leq n$, try setting all levels to the right of r to max capacity and all levels to the left of r to have just under x coins. We also need to determine the index l where all levels to the left of l are at max capacity and levels between l and r have just under x coins. We can maintain l and r with two pointers, leading to an $O(n \log n)$ solution (with binary search to determine x). We can optimize this to $O(n)$ by using a formula to determine x, but this optimization was not necessary to solve the problem.

8 A Nightmare of a Grid

First, let's solve this for $k = 1$. Make the array $ok_{i,j}$ which is 1 if $g_{i,j} = g_{i+1,j} = g_{i,j+1} = g_{i+1,j+1}$, otherwise 0. We can answer queries with 2D prefix sums of ok.

Then extend ok to three dimensions, with $ok_{k,i,j}$ equal to 1 if $g_{i,j} \dots g_{i+k,j+k}$ are the same and 0 otherwise. Note that $ok_{k,i,j} = \min(ok_{k-1,i,j}, ok_{k-1,i+1,j}, ok_{k-1,i,j+1}, ok_{k-1,i+1,j+1})$ for $k > 1$, so we can compute each element of the array in $O(1)$ time, and we can still answer each query in $O(1)$ time. The critical observation is that $k_{max} \leq \min(n, m) \leq \sqrt{nm}$. Our final time (and memory) complexity is $O(nm \cdot min(n, m) + q) \leq O(nm\sqrt{nm} + q)$, which is good enough.

Alternate Solution (faster and less memory)

Use DP. Let $dp_{i,j}$ be the max k s.t. $g_{i,j} \dots g_{i+k,j+k}$ are the same. Note that $dp_{i,j} = \min(dp_{i+1,j}, dp_{i,j+1}, dp_{i+1,j+1}) + 1$ if $g_{i,j} = g_{i+1,j} = g_{i,j+1} = g_{i+1,j+1}$, otherwise 0.

Sort all queries in decreasing k and keep a 2D segment (or fenwick) tree, initially all zeros. Before a query of k, set all values (i, j) where $dp_{i,j} \geq k$ to 1, and do a 2D range sum to answer the query. Our final time complexity is $O((nm + q) \log n \log m + q \log q)$.

9 Custom Complexity

We can use PIE to sum the grid, adding a grid (a), subtracting some "borders" (b), and adding back some "corners" (c). The following figures demonstrate this idea:

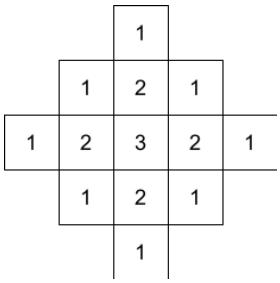

Figure 1: a

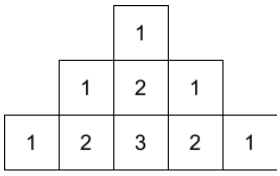

Figure 2: b

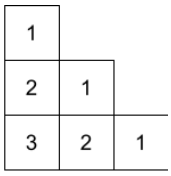

Figure 3: c

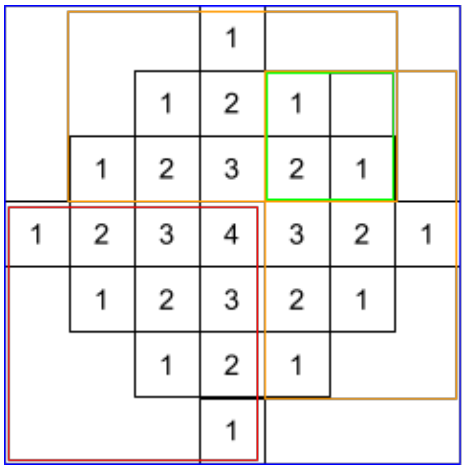

Figure 4: Red: $n \times m$ grid, Blue: (a), Orange: (b), Green: (c)

For each of the sums we need to calculate, we have to sum $O(T^2)$ numbers, each of which is $O(T)$, for each of $O(T)$ distinct times t. Thus, each sum is quartic in T. But instead of trying to derive the formulas for all three grids ourselves, we can just plug in small values to a statistical regression on Desmos, getting the following equations for the xth term in the grids:

Grid (a): $\frac{1}{6}x^4 + \frac{1}{3}x^3 + \frac{1}{3}x^2 + \frac{1}{6}x$

Grid (b): $\frac{1}{12}x^4 + \frac{1}{3}x^3 + \frac{5}{12}x^2 + \frac{1}{6}x$

Grid (c): $\frac{1}{24}x^4 + \frac{1}{4}x^3 + \frac{11}{24}x^2 + \frac{1}{4}x$

All that remains is to implement this logic.

Similar problem: 1822C

10 Fun Addition

Basically, the problem is: Find the sum across all intervals where the two endpoints have the same b_i of the minimum a_i in that range multiplied by the length of the interval.

Fix the value of b_i. Then we can maintain a stack that is monotonically increasing in $\min(a_i)$ to store the sum of all intervals that end at a certain index. We actually need to store a lot of things in this stack in order to maintain the answer properly: the value of $\min(a_i)$, the number of intervals that have that value, the sum of the lengths of the intervals, and the last index i that we processed. In all, this will take $O(n)$ time.

11 Binary Spikes

Since we are subtracting the binary arrays a and b, we can define a new array $v_i = a_i - b_i$ and disregard the arrays a and b. The problem becomes sum of $|\,\text{pathSum}(v, a, b)|$ over all $1 \le a, b \le N$. Now, change v_i to be the sum of v_i from node i to the root (node 1). This means $\text{pathSum}(v, a, b) = v_a + v_b - v_{\text{lca}(a,b)} - v_{\text{parent}(\text{lca}(a,b))}$. Iterate over $\text{lca}(a, b)$. We can use small to large merging to iterate over all nodes a for a specific lca, and find the sum of the absolute value over all nodes b in a different subtree (but with the same lca). This means that $v_a - v_{\text{lca}(a,b)} - v_{\text{parent}(\text{lca}(a,b))}$ is fixed so we can set it to k and find the sum of —k+v[b]— over all b. To make the coding easier and more intuitive (in my opinion) negate k to find sum of $|v_b - k|$. This is negative when $v_b < k$ and positive when $v_b > k$. Note that the minimum value of v_b is $-n$ and maximum is n. This means we find the sum of $(k - v_b)$ when $-n \le v_b < k$ and $(v_b - k)$ when $k < v_b \le n$. Since we removed the absolute value, we can split up the sum into finding the sum of k and sum of v_b. Since k is constant, the sum of k is $k \cdot$ number of $v_b < k$ for the first sum and $k \cdot$ number of $v_b > k$ for the second sum (represented by the variable cnt1). To calculate the sum of v_b, we can just set $arr_i = i \cdot$ number of $v_b =$ current i over all b) and find sums over the array arr (represented by the variable $cnt2$). We can speed up both of these sums using a sparse segtree. Note that every pair of nodes is counted once here, and in the problem they are counted twice (cause of (x, y) and (y, x)) so we multiply by 2 and add the sum of $|v_i|$ for single nodes.

12 Tidal Wave

TL;DR: Simplify to CHT, throw DS at it until it's $O(n \log^2 n)$

First note that for each starting node x, the optimal solution is to travel to some target node i, then take a_i repeatedly (this can be proven with exchange argument). So the answer for node x is

$$\max_{1 \le i \le n} (path(x, i) + a_i \cdot (n - dist(x, i) - 1))$$

, where $path(a, b)$ is the sum of the values on the nodes between a and b, inclusive, and $dist(a, b)$ is the distance on the tree between a and b. Taking this expression directly we can already get an $O(n^2)$ solution with DFS.

You may realize that the score expression is like a linear equation, with the y-intercept being $path(x, i)$ and the slope being a_i. This sounds a lot like convex

hull trick! This inspires us to use LineConTainer (LCT) or Li Chao Tree (LCT) to maintain dynamic CHT. We can then split the computation into two parts:

Case 1: i in the subtree of x

A common trick to solve subtree problems is to take the answers from each subtree of x and merge them to determine the answer for node x. For each node x, create an LCT storing the scores for all nodes in the subtree of x. We can then use small-to-large merging to merge the LCTs of the children of x efficiently.

The above almost works, except the $dist(x, i)$ and $path(x, i)$ values for each node in the line container may change.

To solve the *dist* problem, we can utilize the choice of what x-coordinate to query in the LCT. One way is to always query $depth_x$ (shifting the y-intercepts correspondingly). This works because the queried x-coordinate always shifts left by 1 when we move to the parent, which corresponds to having 1 less turn to take a_i.

To solve the *path* problem, instead of shifting the y-intercept of each node in the LCT up by a_x, we can define an offset variable and "shift" all the lines by adding the offset every time we query the LCT.

In total, this will take $O(n \log^2 n)$, since adding a line to LCT takes $O(\log n)$ time and we do this $O(n \log n)$ times because of small-to-large merging.

Case 2: i outside the subtree of x

We essentially just extend the idea of Case 1. Using HLD, we can group each node i into the subtree of some node along one of $O(\log n)$ heavy paths. Then, we can build an LCT starting from the node on each heavy path closest to the root and moving away from the root. Each time, we use the DFS from Case 1 to add the nodes in the subtree to the LCT. We also store the LCT queries along the way.

Since each node is only added to $O(\log n)$ LCTs (corresponding to each light edge on the path to the root), it will take $O(n \log^2 n)$ to build the LCTs. In addition, we can determine the deepest light child of each node to only store LCT queries that will actually be used, so it will take $O(n \log n) \cdot O(\log n) = O(n \log^2 n)$ time to make all of the queries. Finally, when traversing to the root, each node will only make $O(\log n)$ lookups from the precomputed queries.

Combining the results from these two cases, we get that the final running time is $O(n \log^2 n)$.

13 Punching Wood

Ignoring the number of slots in the inventory, to find the number of filled or partially filled slots, we need to compute $\lceil \frac{n}{64} \rceil$. However, we also have to cap this value at 36 (because there are only 36 slots). Thus, we need to find $\min(36, \lceil \frac{n}{64} \rceil)$.

While implementing, use the identity $\lceil \frac{n}{k} \rceil = \lfloor \frac{n+k-1}{k} \rfloor$. (Floor division is the `//` operator in Python, or `/` in Java/C/C++).

14 Climbing Higher

After doing some analysis, we can conclude that only the largest and second largest elements in the array matter at every instance. We can first use sorting to find the 2 largest elements, then continuiosly perform the adding operation on those 2 elements. You might think that this would time out because you may have to perform the adding operation many times, however it won't. This is because even if you let the 2 largest numbers be 1 and 1, the numbers blow up quickly. It takes just 86 operations for the largest number to exceed 10^{18}, larger than any value of x. Infact, this is the fibonacci sequence, which takes $O(\log(N))$ operations to exceed a given value of N.

15 Crop Optimization

Let's put all a_i+a_{i+1} in a multiset, so that we can easily query the max value, i.e. the badness. For each i, let's try removing that element, and see how it changes the badness. We remove value $a_{i-1}+a_i$ and a_i+a_{i+1}, and add value $a_{i-1}+a_{i+1}$ (ignoring out of bounds elements). We can temporarily remove/add these values from the multiset, adding/removing them back in/out after processing the value i. After doing this, query the maximum value in the multiset, representing the badness. If we do this over all i from 1 to N, and find the minimum, that is the least badness of the array so we can return that minimum value (minimum over all maximums of the multiset).

16 Redstone Subtraction

Note it is not possible to make a_1 negative since there is no $i < 1$. For each i from 2 to N, let's try computing the number operations needed to make that element negative. Let m be the maximum value from $a_{1\ldots i-1}$, do the operation on that value as many times as we need to. This means we need $v_i := \lceil \frac{a_i+1}{m} \rceil$ operations to make a_i negative. After getting the minimum number of operations for each element, we have to find the minimum total number of operations for each k from 1 to $N-1$. To do this, we can sort the array v, and compute the prefix sum for each value of k, and print each one.

17 Storage Sync

Let's look at the operations more closely. Is it possible to swap 2 elements in an array? If it is possible, we can use multiple of these swap operations to sort each array individually. The answer is yes! Let's try to swap the first and third elements in these 2 arrays in the following pair of arrays:

Initial:
1 2 3 4
5 6 7 8

Operation 1: 2 3
1 2 7 4
5 6 3 8
Operation 2: 1 A 2
7 4 1 2
5 6 3 8
Operation 3: 2 3
7 4 3 2
5 6 1 8
Operation 4: 1 A 2
3 2 7 4
5 6 1 8
Operation 5: 2 3
3 2 1 4
5 6 7 8

Using a sequence of 5 operations, we can swap any 2 elements in an array. We can sort an array by swapping the smallest element to the start, second smallest to the second, and so on for the entire array. This means that the array can be sorted in N swaps, taking a total of $5N$ operations. Using the same idea for the second array, it takes a total of $10N$, well under the $60N$ limit."

18 Beacon Foundation

First consider the mod condition, which reduces to the following: When the numbers are sorted in decreasing order, $s_{i+1} \mid s_i$ for all $1 \leq i < k$. This means that if $k > \log_2(n)$, the answer is -1. (This is also why there is no limit on the sum of k over all test cases).

Otherwise, we are looking for the lexicographically largest sequence when sorted in decreasing order. This means firstly that we want to maximize the largest element in the sequence, so we need to find the largest number less than n such that its "chain" can continue for long enough. It turns out this is equivalent to the sum of the powers in its prime factorization being at least $k - 1$. We can use a sieve to precompute the number of prime powers in each number from 1 to 10^6, and there are many ways to search for the largest number afterwards, for example using offline queries and binary search.

Once we get the largest number in the sequence, we have to repeatedly divide by the smallest factor until we hit 1 (since we are looking for lexicographically maximum).

19 Draining Lake (lake)

$O\left(N^5\right)$ solution:

Start with a N by N grid, where each cell denotes if it is possible for the drain to exist there. Now, for every query, we ideally want to split the possible region in half. We can enumerate over all possible guesses for each turn and take the one that splits the possible region as close to half. This takes $O\left(N^5\right)$ time.

$O\left(N^3\right)$ solution:

One observation is that it only takes $\log\left(N^2\right)$ seconds to find the drain if each guess perfectly splits the area in half. This means that as N increases you can be less precise with each guess. With this we can modify our strategy. Instead of enumerating over all possible guesses, instead we will make random guesses for the first $N-10$ seconds. Then, use the enumeration algorithm for the last 10 seconds in order to make sure it is narrowed down to 1 drain location.

For the random guesses we store an array of all possible drain locations and get rid of a location if it contradicts our guess. This takes $O\left(N^3\right)$ time because there are N^2 locations and N guesses. The final 10 guesses use the enumeration algorithm, however it is much faster than $O\left(N^4\right)$ due to there being only a few valid drain locations left (around N).

The problem has a lower bound for N at 10 because for $2 \leq N \leq 9$, the problem is impossible. For the first few seconds, your ability to disqualify possible drain locations is bounded by the size of the region you can query in a second. After a certain time, the bound switches to be half of the possible region size.

For example, for $N = 9$, we have:

0 seconds: 80 possible drain locations (you can only query 1 square)

1 second: 75 possible drain locations

2 seconds: 62 possible drain locations

3 seconds: 37 possible drain locations

4 seconds: 19 possible drain locations (your query region is large enough to cut it in half)

5 seconds: 9 possible drain locations

6 seconds: 5 possible drain locations

7 seconds: 3 possible drain locations

8 seconds: 2 possible drain locations

It can similarly be proven that for $2 \leq N \leq 9$, the problem is impossible.

20 Mirror Match

To proceed with the solution, we first have to figure out how to keep track of the array, and the posititions of each number in the array, after doing an operation. We can do that with the following code:

```
function<void(int,int)> doSwap = [&](int i, int j){
if(i == j) return;
swap(p[i],p[j]);
swap(a[p[i]],a[p[j]]);
swap(p[n-1-i],p[n-1-j]);
swap(a[p[n-1-i]],a[p[n-1-j]]);
```

ops.push_back({i+1,j+1});
};

where p is the permutation and a are the positions of the elements in the permutation. *ops* is the final operations array that we will print out at the end.

Let's first find out when it is possible to sort. If n is odd, a 3-cycle of any triple of elements is possible. You can show this by showing it is true for all odd $n \leq 7$. The proof to extend this to all odd n is left as an exercise to the reader. For even n, it is possible if $p_i + p_{n+1-i} = n + 1$ over all i, and the part about 3 cycles is true. This is because when n is even, p_i and p_{n+1-i} will always be "paired" and can't be seperated.

The model solution sorts the array in at most $\frac{3n}{2}$ moves, however any solution that takes up to $2n$ moves still passes. To sort the array, we can first put 1 and n in their place, then go inward putting 2 and $n-1$ in their place, and so on. When n is even, we can put them in their place using at most 1 swap operation, making the answer at most $\frac{n}{2}$ when n is even. When n is odd, we can put them in their place by moving 1 to the center, swapping 1 and n, (putting them at opposite positions) and then put them in their spot in 1 move. Note that the moves aren't always this simple, sometimes you will have to do some casework if 1 and n are in weird positions (like already in the middle or swapped in positions). After doing this, the center element should be sorted by pigeon hole. Since there are $\frac{n}{2}$ pairs, and each pair takes at most 3 operations to sort, for odd n it takes at most $\frac{3n}{2}$ operations to sort the array.

21 Minecart Mania

Reading the problem, we can quickly come up with a solution in $O(n^3)$: For each possible pair (i, j), try creating a portal between i and j and check how long it takes to move each number to its destination.

To speed up this solution, let's fix i and calculate the answer for each possible j. For each i, call $\text{cost}_{k,j}$ the contribution of number k to the total cost if we put the other end of the portal at j.

Looking at the difference array of cost_k (e.g. $\text{cost}_{k,j} - \text{cost}_{k,j-1}$), we can observe that the difference array will be some number of 0's (corresponding to the portal not being worth using), followed by some number of -1's (corresponding to the portal getting closer to the start/endpoint), followed by some number of 1's (corresponding to the portal getting farther away from the start/endpoint).

Thus, the difference array of the difference array (AKA the 2nd derivative) of cost_k with respect to j will only have $O(1)$ nonzero elements. Summing the 2nd derivative of cost over all k, we can calculate the answer for all j over fixed i in $O(n)$. Since there are n possible values of i, our final time complexity is $O(n^2)$.

When implementing, make sure to try both possible ways of moving through the portal (a bit like Teleportation).

22 Ender Chest Secrets

We will solve this problem by computing over all subarrays, $\sum_{i=1}^{n} i \cdot f(i) \cdot k!$ where $f(i)$ is the number of permutations where i is the subarray maximum and k is the number of -1s in the subarray.

Let L_i and G_i be number of values $< i$ and $> i$ which do not exist in p, respectively. Denote m as the existing maximum for a given subarray.

For all $i > m$, $f(i) = \binom{G_i}{k-1}$ since one of the -1s must be i. We can precompute the suffix sums $\sum_{i=m+1}^{n} \binom{G_i}{k-1}$, for a fixed m and k in $O(n^2 \log n)$ beforehand, and the transition from $i = m + 2$ to $i = m + 1$ will be $O(\log n)$.

For $i = m$, $f(i) = \binom{L_i}{k}$.

For $i < m$, $f(i) = 0$.

Be sure to first precompute all factorials from 1 to n in $O(n \log n)$ and carefully handle all operations modulo $10^9 + 7$.

Time complexity: $O(n^2 \log n)$, space complexity: $O(n^2)$.

23 Redstone Cascade

Consider a tree with 3 nodes where 3 is the child of 2 and 2 is the child of 1. Let's try to find the probability of node 3 happening assuming node 1 hapens, and 3 happening of node 1 dosen't happen. The probability that nodes 1 2 and 3 happen is $a_2 \cdot a_3$. The probability that nodes 1 and 3 happen but not 2 is $(1 - a_2) \cdot b_3$. Following this logic, we can conclude that given a and b for nodes 2 and 3, we can find the values of a and b on the transition from node 1 to 3. Using this logic, we can find the values of a and b from one node to any of the ancestors of that node. This can be sped up using binary jumping.

When making this problem, I (Rishikesh) didn't know what a virtual tree was, so I derived it myself to make the problem. Now that I know what it is, I can just tell you to search up what it is if you don't know already.

For each query, construct a virtual tree for that query. To find the values of a and b for each edge, use the binary jumping strategy from before. Now, the problem turns into finding the probability that all leaves in a tree are activated. This can be done easily using dfs, and accounting for the probability that the top node is happening using the combination strategy from before.

24 Endgame Tree

Brute force: $O\left(N^2 Q\right)$

Find all path xor's using dfs, and find the MST using your favorite MST algorithm (the code uses prim's).

Slow Solution: $O\left(NQ\right)$

Note that due to the properties of xor, all graph edge values will be either 0 or 1. Consider each graph edge of weight 0 to be connected, then the answer is the number of connected components-1. Let v represent the array of values so $v_i \in \{0, 1\}$ by definition.

Case 1: There exists a tree edge with both sides being 1.

Let a and b be the nodes on either side of the tree edge. Since $v_a \oplus v_b = 0$, a and b are connected. All other nodes are connected to either a or b because $\text{pathXor}(i, a) \neq \text{pathXor}(i, b)$ because $v_a = v_b = 1$, therefore by pigeonhole all nodes are connected and the answer is 0.

Note that if you have 2 $0s$ connected by a tree edge, you can merge them into the same 0 and merging their edges into the same 0. This means after this process there are no adjacent $0s$, meaning that the values in the tree are alternating since there are no adjacent $1s$. We will be working on this alternating tree from now on.

Case 2: $\sum v_i = 0$

If all v_i are 0 that means every edge in the graph is 0 meaning the answer is 0.

Case 3: $\sum v_i = 1$

This means there is a singular node with value 1. Every node on each side of the 1 must pass through exactly 1 value 1 to get to another node in the tree, meaning that they aren't connected in the graph. This means the number of connected components is 1 more than the degree of the node because the node is in a connected component by itself, meaning the answer is the degree of that node.

Case 4: There exists a node with value 0 that has a tree edge to every node with value 1 in the tree.

The node with value 0 in the middle cannot be connected to any other node in the graph because it must pass through exactly 1 value 1 to get to any other node. This also means that every other node passes through exactly 0 or 2 value 1 nodes to get to every other node in the graph, which means they are all connected to each other. Since there are 2 connected components, the answer is 1.

Case 5: All other cases

First note that each node with value 1 is directly or indirectly connected to every other node with value 1 because if they can connect with xor 0, they are connected and if they can't, there is a node with value 1 in between them and both the 1 values will be connected to the middle 1. Every node 0 is can be connected to a node with value 1. For every node 0, there exists a path starting at that node that contains 2 values 1 in the tree, because if there isn't that means that node 0 is connected to every node 1. This means that path has xor 0 and is connected. This means the answer is 0.

We can easily get an $O(n^2)$ solution from the above: Case 1 is easy to find by iterating through all edges. Case 2 and 3 are also easy by finding the sum of v. To differentiate between case 4 and 5, we can check if there is a connected component in the tree that has all zeros, and is touching every one. We can do

this using dfs and counting the number of ones the component touches. Run this algorithm after each query to get the answer.

Fast solution: $O((n+q)\log n)$:

Cases 2 and 3 are easy to keep track of throughout queries by maintaining the sum of ones.

The trivial solution to keeping track of case 1 however is $O(degree)$ of the node however, and this is too slow. To speed this up, we can keep track of the parent of each node in an array called up and the number of children that are ones in an array called *oneChild*. Also keep track of the total number of edges where both sides of the edge is a one, and update all the arrays after each query. This can be done in $O(1)$ time per query.

Differentiating between cases 4 and 5:

This is equivalent to asking if there exists a path in the tree with a sum of 3. This is far trickier than the other cases.

Credit to Sujay Konda for the following explanation

There are multiple ways of doing this. If I'm not mistaken, there is a $O(n\log^2 n)$ way of maintain the max # of 1s in a path using HLD, but this solution is much cleaner.

Hint 1: What if we rooted the tree at a node with value 1.

Observation 1: Notice that if a 3-path exists, you can force the 3-path to use the root instead.

Proof 1: We will call the three nodes with value 1 in a 3-path the "start", "middle", and "end". Since all nodes on the simple path from the "start" to the "end" are ancestors of either the "start" or the "end", the "middle" has to be an ancestor of the "start" or the "end". Whichever it is, you can go from the root to the "middle" to whichever one the "middle" is an ancestor of, creating a 3-path which uses the root.

With the previous hint, we know if a 3-path exists, there exists a 3-path from any node with value 1. So now we can pick some node with value 1 and check if it has 3-path. This results into two cases, either that node is a "middle" of some 3-path or the "start/end" of some 3-path.

Hint 2: How many nodes with value 1 only have 3-paths where they are the middle?

Observation 2: Atmost 1.

Proof 2: If there are two "middle" nodes, then root the tree at one of the middle nodes. Using the same argument in Proof 1, we can show there exists a 3-path from this node to the second "middle" node which will make this not actually a "middle" node since it would be a "start" of a 3-path in this case.

This basically means you can check two different nodes with value 1 and see if they have a 3-path starting from there and ending elsewhere. Let sm_i be the sum of the values of the ancestors of node u. ¡br¿ Suppose we are checking node u. To check for a 3-path, we want to find $max(sm_u + sm_v - sm_{lca(u,v)} + v_{lca(u,v)})$ and see if it is greater than 3. We can use a segment tree on the Euler Tree Tour to store sm_i for all i. Then by picking the two nodes with least depth as the ones to check, since there are very little nodes with value 1 above those either them. In fact, the only case is the first one being above the second one, so we can ensure

that $v_{lca(u,v)} - sm_{lca(u,v)}$ turns to 1 if $lca(u, v)$ is below the first node (when checking the second node). There is also the case when $lca(u, v) = u$, because $v_u = 1$. You can handle both the cases within the euler tour, while maintaining $max(sm_v - sm_{lca(u,v)} + v_{lca(u,v)})$ (either with range updates or with a segtree storing prefix maximums). Refer to the code for specific implementation details.

So now we can root it at a one node and we want to see if there exists a path ending at that node with sum 3. To do this we can use euler tour. The issue is euler tour only works by assigning weights to children so basically if you assign a 1 to a node you assign 1 to the edge upwards of that node. then if you followed the euler tour going from one node to the other you would get the sum of the path (or negative the sum of the path since going down an edge would be negative and going up an edge is positive). But there is also this issue of you miss the lca and since not all the paths are just going down and just going up you might get wrong values. But none of this matters if you pick the node with one closest to the root. But since you pick two nodes, you pick the two closest its a bit annoying for the second one if its underneath the first one. Though technically it must follow that there exists a second one which has no ones above it. So maybe u could find that one.