Teamscode Summer 2025

omeganot

August 2025

1. Squares - Novice A / Advanced A

   Since everything in the input is a square, if we draw a square that covers all of the input squares, the intersections of each input square and the one we drew will be a square. Thus, we can draw a square with bottom left corner at $(-10^9, -10^9)$ and top right corner at $(10^9, 10^9)$. This solves the problem in $O(N)$.

   Code

2. Bocchi and the Neural Network

We can take advantage of the fact that each value only appears once. Thus, if we want our MEX to be greater than 1 (it will be at least 1 since we start at 0), we must visit the 1 node on its layer, and thus cannot include any node in 1's layer in our MEX.

Similarly, if we want our MEX to be greater than 2, then we have to ensure it's not on the same layer as 1. If we want our MEX to be greater than 3, it can't be on the same layer as 2 or 1, and so on. Thus, we can record which layer each node is on, and then iterate from 1 to n, checking if the next node is not in a layer which is already 'used up' by our earlier nodes. We can record whether or not a layer has been used up in a boolean array. Thus, we solve the problem in $O(n)$.

Code

3. Snowing - Novice C

We want to reinforce the k nodes that break the earliest. Thus, we must calculate when each node will break without reinforcements.

Each subtree increases by a fixed weight each day, so we should calculate the increase of a subtree for each day. That is simply the sum of all a_i in the subtree. This can be calculated through a DFS. Let the sum of a_i of node x's subtree be s_x. Then, we can figure out how many days of snow will be on the tree before the tree breaks, which is $\lfloor \frac{w_x}{s_x} \rfloor$.

If we sort the nodes by the number of days they can withstand from smallest to largest, then we know that for each k, the answer is the number of days we can withstand (after reinforcing the k weakest nodes) multiplied by s_1, the total number of snow piled on each day. Thus, due to sorting, we solve the problem in $O(n \log n)$.

Code

4. Sum and Or - Novice D

Since Alice goes first, she can predict the end result of picking index i, as she knows that Bob will then try to pick j to maximize the result. Thus, we want to find the i that minimizes the result, knowing what Bob will do. It's a bit too slow to do this naively, by iterating over all i then trying j to find the maximum, as that takes $O(n^2)$.

Since Alice can only change one element of A, for all the unchanged elements we can precompute how much the result would increase by if Bob selects that index. We want the maximum of all those other indices, and can compare it to Bob picking $j = i$ himself, taking the maximum over the two. We want to minimum over all i.

We can do this by finding the two j with the largest value of $A_j|y - A_j$. We know that the maximum of all the indices aside from i will be one of these two. Thus, when checking i, if the j that maximizes $A_j|y - A_j$ equals i, then we take the second largest. We solve the problem in $O(n)$. Code

5. Trolley Problem - Novice E

We can reverse array t in $O(n)$ so that t_1 is the first trolley and $t_1 \geq t_2 \geq \cdots \geq t_n$. Since the trolleys go in order of heaviest to lightest, if one trolley leaves a cow sleeping, then all other trolleys will be able to not scare off any cows since we can send those trolleys down the same lane.

Thus, we want to know which is the earliest trolley with which we can not scare off any cows. Our answer is then the number of trolleys that go before it. Consider some trolley i. If we want trolley i to not scare off any cows, we know there are $i-1$ trolleys that went before it who scared off cows. Thus, if $\max(c_1, c_2, \ldots, c_i, d_1, d_2, \ldots, d_i) \geq t_i$ then we know that at most we will have to scare off $i-1$ cows.

Let's say we want trolley i to run into cow c_j. Then, we should let trolleys $1 \ldots j$ scare off all cows from $c_1 \ldots c_{j-1}$ then send the rest of the trolleys down the d lane. Thus, we can then block trolleys $i \ldots n$ with cow c_j. This logic applies to wanting to collide trolley i with cow d_j. This is why it is sufficient to just compare $\max(c_1, c_2, \ldots, c_i, d_1, d_2, \ldots, d_i) \geq t_i$.

Thus, we can iterate over all trolleys and maintain $\max(c_1, c_2, \ldots, c_i, d_1, d_2, \ldots, d_i)$, checking if this value is greater than or equal to t_i. If we ever run out of cows in a lane, then we also know the rest of the trolleys can be sent down that lane and not scare any cows. We have solved the problem in $O(n)$.

Code

6. 345 - Novice F

Let the given array of 5 numbers be a.

We may observe that every operation involves a_3. Furthermore, almost all operations involve a_2 and a_4. Only decreasing the three left elements doesn't impact the a_4, and decreasing the three right elements doesn't change a_2. Thus, the number of times we decrease the three left numbers must be $a_3 - a_4$, and the number of times we decrease the three right numbers is $a_3 - a_2$.

Since we know these operations must happen, we can apply them and consider our new array where we can do 3 operations: decrease the middle three, left four, or right four. We can see now that to decrease the a_1 we must decrease the left four elements, and to decrease a_5 we must decrease the right four elements.

Thus, now we know that the new a_0 is the number of times we decrease the left four elements, and the new a_5 is the number of times we decrease the right four elements. Then, the a_3 that we are left with is the number of times we decrease the middle three elements.

This shows that there is at most one way to decrease all elements of a, so it will be minimal. If at any time our method finds negative numbers, then we know it's impossible. We have thus solved the problem in $O(1)$.

Code

7. Max Binary Tree Width - Novice G / Advanced B

Although it seems hard to calculate the maximum width given nodes, it is a bit easier to calculate the minimum number of nodes needed given a width. Let's call this function $f(x)$ (x is the width needed, $f(x)$ the minimum number of nodes needed) .

$f(x)$ is clearly monotonically increasing, so we can binary search on the answer! How do we calculate $f(x)$? Well, if we know the width of the tree is x, then the last level should have x nodes (if the last level doesn't have x nodes, then we can delete that level and our tree still has a width of x since some level must have x nodes).

Thus, to construct a tree with a width of x, we also need a tree of width $\lceil \frac{x}{2} \rceil$ (ceiling since if we have $x = 3$, the previous level must have $\lceil \frac{3}{2} \rceil = 2$ nodes), and then add our x nodes below that level.

We can calculate f recursively. $f(1) = 1$, and for $x > 1$ $f(x) = f(\lceil \frac{x}{2} \rceil) + x$. Since x is halved each time, the time complexity of calculating $f(x)$ is $O(\log x)$. Since we are binary searching, we call $f(x)$ $O(\log n)$ times. Thus, we solve the problem in $O(\log^2 n)$. Since there are only $2 \cdot 10^5$ test cases with $n = 10^9$, we are able to solve the problem comfortably.

Code

8. Trivial Problem - Novice H / Advanced C

We may notice that f is a monotonic nonincreasing function, as if $i < j$ we know $f(i) \geq f(j)$. This is because all elements of a which are the maximum value for some subarray of length j must also be the maximum value for some subarray of length i, so the mex can't increase from i to j.

As such, we are motivated to iterate from $k = n$ to $k = 1$, and as we decrease k we add new maximums of subarrays to our set while maintaining the mex. This can be done by keeping track of a set of all of our maxes, and after we insert our new maxes in, we increase our current mex while it is in the set. This amortizes to $O(n \log n)$ since we only perform $O(n)$ set insertions and our mex can only increase $O(n)$ times.

We can actually bring this down to $O(n)$ by maintaining a boolean array of length n that keeps track of which elements are in the set instead of using an actual set.

All that remains is to find which new maximums of subarrays are present as we go from $k + 1$ to k. As such, we are interested in the maximum k for each a_i such that a_i is the maximum of a subarray of length k. We can store in a vector for each k which a_i have the maximum length (of a subarray with a_i as the max) equal to k.

To find this, we need to know the nearest larger element to the left and the nearest larger element to the right. Let those values be l_i and r_i, respectively (If the nearest greater element to the left doesn't exist, $l_i = 0$, and similarly $r_i = n + 1$). Then, the maximum k would be $r_i - l_i - 1$. We can calculate l and r in $O(n)$ with a Monotonic Stack. Thus, we can solve the problem in $O(n)$.

Code

9. Pennant Hanging - Novice I / Advanced D

I argued to remove L from the problem as it doesn't really matter but I was vetoed sadly.

As with all query problems, our first step is to solve the problem without queries. Given a pennants of rank k, we would have to move all the pennants with rank greater than k over to the right by a units. Note that all the moved pennants are on a suffix of r.

However, this doesn't mean we must move each pennant with $r_i > k$. If $r_i = r_{i+a}$, then we don't have to move pennant $i + a$. Furthermore, we must move pennant $i + a$ if $r_i \neq r_{i+a}$, so by moving only these pennants we achieve the minimum answer.

Thus, given a query (a, k), we want to count the number of pennants with $r_i \neq r_{i+a}$. It's not too difficult to find the suffix of pennants we must move, since we can use binary search, but it's hard because we'd have to count how many pennants have $r_i \neq r_{i+a}$, which is a bit harder.

Since all pennants of a similar rank are placed next to each other, checking the condition $r_i \neq r_{i+a}$ is the same as checking the condition $b_i > i + a$, where b_i is the nearest different pennant to the right. $b_i > i + a$ can be rearranged into $b_i - i > a$. Thus, for a given suffix (defined by k) and an integer a we want to count how many indices on this suffix have $b_i - i > a$. This is essentially a "count number of elements less than x on a suffix" problem, which we can solve offline. We can calculate $b_i - i$ in $O(N)$, though I think you are able to figure that out (plus, I am lazy).

We will iterate over suffixes from the smallest suffix to the biggest suffix, so from right to left. We can maintain some sort of order statistics data structure like an Ordered Statistics Tree or Segment Tree with range sum queries and point updates. I use a segment tree in my code.

When we extend our suffix, we just need to add insert $b_i - i$ into our set. Then, queries can be answered with our chosen data structure. In my code, I sort the queries by k from greatest to smallest and maintain the current suffix to answer the queries appropriately. The sorting takes $O(Q \log Q)$ and our segment tree operations are $O((N + Q) \log N)$ in total (since for each suffix we perform one update and for each query we perform one query), meaing we solve the problem in $O(Q \log Q + (N + Q) \log N)$

Code

10. Castlefall - Novice J / Advanced E

A pair of words can determine two teams if each player can have at least one of the two words and, for each of the two words, at least one player has the word. In set terminology, if we imagine each word having a set of players who can have the word, a pair of words can split two teams if the union of the sets is all N players, and no set is empty.

Since there are very few players, we can use bitmasks to represent each word. Let a_i be the mask of the ith word. The jth bit of a_i is set if player j can have word i, and 0 otherwise. If we can find more than one or zero pairs of (i, j) such that $a_i|a_j$ has N set bits (union of both sets is all N players) and $a_i, a_j \neq 0$.

For word i to have a potential pair with j, we already know $a_i|a_j$ has N bits. Thus, $(2^N - 1) \oplus a_i$ (all the off bits in a_i) must be a subset of a_j. We can use SOS DP to count the number of j (for each possible mask of a_i) that satisfy this condition as well as keep track of a valid j in $O(N2^N)$.

Now, just because there is only one pair (i, j) doesn't mean there is also only one way to create teams. If a_i and a_j share a bit then that bit could go in either team. There are three cases, the first being that the bit is the only set bit in a_i or a_j, meaning that bit has to go to that team, leaving all the other bits for the other team. The other case is when this is not true but $a_i \& a_j \neq 0$, then there will definitely be more than one way to make teams. The last is when

The way I handle this case is by checking if there are any words with all bits set, since if a pair (i, j) are in case 1 then either a_i or a_j must have all bits set if $a_i|a_j$ has all bits set and $a_i \& a_j \neq 0$. If there exists a word with all bits set, then there can only be one other word with any bits set, and that word must have only 1 bit set. These conditions can be checked in $O(N)$.

To handle the other cases (now assume that no word has all bits set), we can iterate over all words i and use our SOS DP precomputation to see if valid j exists. If it does, we must ensure $a_i \& a_j = 0$. If we only find one (unordered) pair of (i, j), then we have our answer. If we find no pairs or more than one, the answer is no. To calculate each mask a_i takes $O(NM)$, so our total complexity combined with the SOS DP is $O(NM + N2^N)$, and we have thus solved the problem.

Code

11. Graph Problem - Novice K / Advanced F

Let g be the greatest common factor of k and all the edge weights. Intuitively, all the valid residues we can obtain are x such that $0 < x < k$ and x is a multiple of g. The natural question is, are all multiples of g achievable?

Bezout's Identity can be generalized to multiple integers, if we imagine $e_1, e_2, \ldots, e_m$ as the array of all edge weights, there exists $c_1, c_2, \ldots, c_n$ such that $\sum e_i c_i = g$. If we take all integers modulo k, then we have $\sum e_i c_i \equiv g \pmod{k}$.

However, this does not mean that there exists a path from s to t satisfying our conditions. Since we know all weights are a multiple of g and k is also a multiple of g, we can divide k by g and divide all edge weights by g. The answer to these new weights is intuitively the same as the old problem (as we just multiply by g to get the new problem again). From now on, when we refer to k or edge weights, we assume we have already divided them by g.

Now, we ask if we can find a path from s to t for each residue modulo k. We know there still exists $\sum e_i c_i \equiv g \pmod{k}$, since g is now 1. We wonder if we can find a path from s to t that traverses each edge $c_i \pmod{k}$ times. This means that we have to find a way to travel from a to b with the sum of weights being 0 modulo k.

If k is odd, we actually can. Let's say a and b are adjacent. We can then just go back and forth k times, and we will have gone from a to b while the edge is traversed k times, effectively not changing the sum of the path modulo k. Thus, if we start at s, we can visit all edges that we need to and traverse their edge c_i times, then use our method to travel from a to b without changing the sum to get the rest of the edges. We can then go to t without changing the sum, giving us a way to travel from s to t with sum 1 modulo k.

This is great, but can we do all k possible sums? Well, if we can travel from s to t with sum 1, then we just need to go from t to s with sum 0 (which we already showed we can do), then repeat our process! Thus, if k is odd, our answer is k (remember that we already divided the original k by g, so technically $\frac{k}{g}$.

What about for even k? The previous idea doesn't work because if we go back and forth from adjacent a, b k times, we will end up back where we started. So, we can't definitively say there's a way to find a path of sum 0 $\pmod{k}$ from a to b.

If we go back and forth on an edge with weight w, we can have a sum that's a multiple of $2w$ when going from one endpoint to the other. As it turns, out, we can definitely get at least $\frac{k}{2}$ residues from s to t. Let's look back at $\sum c_i e_i$. We can start from s, go to an endpoint of the first edge, then cross that edge twice to get back to the endpoint we started at. We can then travel to the next edge, and do the same, then wehn we're done with the edges return to t.

Our sum is effectively the sum of all the edges we used to go from s to t plus $2\sum c_i e_i$. Since there exists c such that $\sum c_i e_i \equiv x \pmod{k}$ for all $0 \leq x < k$, there exists c such that $2\sum c_i e_i \equiv x \pmod{k}$ for all even x. Thus, if the sum of the other edges from s to t are odd, we can get all odd residues, and otherwise we can achieve all even residues.

If there is a way to do this both with an odd path from s to t and an even path from s to t, then we can achieve all k residues. Though this seems difficult to check (not really sure how best to motivate this), this is equivalent to checking if the graph can be colored with 2 colors such that all paths between two vertices of the same color are even sum, and all paths between two vertices of different colors are

odd sum. Note that this is not a bipartite coloring, just a coloring of the graph with 2 colors (adjacent nodes are the the same color if the edge between them is of even weight).

If we can color the graph, then all paths from s to t will be all odd or all even, and we have already shown we can achieve all residues of a certain parity, meaning the answer is bounded from above and below by $\frac{k}{2}$ (remember, we divided by g, so $\frac{k}{2g}$ is our answer).

If the graph is not colorable like this, then as a baseline we know that we can achieve all residues of some parity. We can find a path from t to itself which has odd sum, since if the graph is not colorable there exists at least one vertex u such that there exists an even and an odd path from u to t (otherwise, u would be a fixed color). As such, we can go from t to u with an even path, then return with an odd path, allowing us to reach all k residues.

To check if the graph is colorable, we take the parity of each edge modulo 2 (another reminder that we have divided edge weights by g). We can then run a DFS, and if the edge from our current node to an adjacent node is 0, they must be the same color. If it is 1, they must be different color. If we ever have to contradict ourselves then the graph is certainly not colorable (we have found an even sum path and an odd sum path from the initial node to this contradictory node).

Thus, to recap all the work we have done, if $\frac{k}{g}$ is odd, our answer is k. If it is even and our DFS finds a valid coloring of the graph, our answer is $\frac{k}{2g}$. Otherwise, if k is even and there is no coloring, our answer is still k. If we submit this, we might see we get wrong answer. That's because if $g = k$, then all edges are a multiple of k and our algorithm finds the answer to be $\frac{1}{2}$. In this case, we must output 1.

Since we perform m calls to gcd with values $\leq w$ (where w is the max weight of an edge, 10^9) (the k doesn't really count as we can argue $\gcd(k, w)$ takes $O(\log w)$), and the DFS is $O(n)$, we can solve the problem in $O(n + m \log w)$.

Code

12. Self Destructing Sokoban Swarm - Novice L / Advanced G

We number the rows from top to bottom $1 \dots n$, and columns from left to right $1 \dots m$. For each cell labelled A, we know it takes one robot to move a robot to that cell. Let the desired cell we want to reach be (x, y).

To be able to push a robot onto a cell (i, j), there are two cases. The first is that (i, j) can spawn robots itself. Otherwise, we will have to push a robot into (i, j). There are four directions that the robot could have been pushed to end up in (i, j), and as such 4 cases:

We can push robots into cells $(i-2, j)$ and $(i-1, j)$, so if we push a robot into $(i-2, j)$ and $(i-1, j)$ we use the robot in cell $(i-2, j)$ to push $(i-1, j)$ into (i, j). We can push robots into cells $(i+2, j)$ and $(i+1, j)$, so if we push a robot into $(i+2, j)$ and $(i+1, j)$ we use the robot in cell $(i+2, j)$ to push $(i+1, j)$ into (i, j). You can probably see how similar logic would work for $(i, j-2), (i, j-1)$ and $(i, j+2), (i, j+1)$.

Although we don't exactly have a graph problem, it feels right to treat the number of robots required to move a robot to cell (i, j) as its distance, so let's do that.

We can use a Dijkstra-inspired algorithm by beginning with a set of nodes which we know the minimum distance to (all cells with an A we know have distance 1), and slowly include the cells with shortest distance that aren't in our set yet until there are no more cells to add or we find our answer.

As such, we will initially insert all A cells into a min priority queue. Then, we take the top of the priority queue and find all new cells that we can visit. There are two cases, either the current cell is the one being pushed, or the one that does the pushing. Regardless, the 'partner' cell that will assist our current cell in pushing robots to unvisited cells must be adjacent to our current cell. One cell will be the pusher, and the other will be the pushed robot.

We can check all 4 directions to see if there is a visited cell directly adjacent, and then try both cases of the current cell being the pusher and the current cell being the pushed robot to see if there are new cells to visit. We call these two cells the "pushing" cells. The pushing cells can push a robot into two possibly new cells, so we must insert these into the priority queue alongside the distance if unvisited. The new distance will be the sum of the distances to the two pushing cells, since only after we push a robot into both original cells can we finally push a robot into the desired cell. We must make sure to relax the distances like we do in Dijkstra's algorithm. Of course, we don't visit wall cells.

Is this algorithm correct? Well, we want to be sure that we visit all the cells possible and have calculated the minimum distance for each. We start with the initial set of all A cells. We know we have found all cells with distance 1. All cells we can visit with distance 2 must result from a pair of cells in this set (note that the distance to the newly visited cell is always greater than the distances to the pushed cells, just like in how Dijkstra there are no negative weights). Intuitively, for each cell with distance 2, our algorithm will find it since the two pushing cells required are both of smaller distance. Thus, our algorithm definitely finds all cells with distance ≤ 2.

We can continue this inductive reasoning infinitely to declare that our algorithm does find the minimum distance to each unvisited cell. Since there are nm cells and a constant number of edges from each cell (all 4 directions and then 2 cells for each direction), there are $O(nm)$ edges, meaning our Time Complexity is $O(nm \log(nm))$.

Code

13. Airplane - Quantum Field Theory Edition - Advanced H

The path which Bessie takes will definitely take the following form: First, Bessie flies from city 1 to some other city $1 \leq i \leq n$ (possibly still 1). Then, Bessie must quantum tunnel from city i to some city $n < j \leq 2n$. Lastly, Bessie travels from city j to city $2n$ ($j = 2n$ is possible).

To optimize the first portion of her journey, Bessie should take the shortest path from 1 to i. Similarly, to optimize the last leg of her journey, Bessie should take the shortest path from j to n. We can compute the shortest paths from city 1 and city $2n$ on the weighted graph by using Dijkstra's algorithm twice, which takes $O(m \log n)$.

Let $a_{i,j}$ be the length of the shortest path from city i to city j. Our answer is the minimum value of $a_{1,i} + d((x_i, y_i), (x_j, y_j)) + a_{j,2n}$. To naively check every pair (i, j) would take $O(n^2)$, so we must find a more efficient approach.

Frequently, when approached with problems involving points on a 2D grid, we can try to use the sweep line technique. We process all the points in order from smallest x to largest x and see if this confers any advantages to us.

As we sweep from left to right, if we encounter one of Farmer John's cities (call it i), we want to know the quickest way to get from i to $2n$, since $a_{1,i}$ is fixed. Since $d((x_i, y_i), (x_j, y_j))$ is the Manhattan distance from i to j, a common trick when finding the minimum/maximum Manhattan distance is to casework on the absolute value: We have

$$d((x_i, y_i), (x_j, y_j)) = |x_i - x_j| + |y_i - y_j|$$

There are 4 cases in total:

$$x_i - x_j + y_i - y_j$$
$$x_i - x_j + y_j - y_i$$
$$x_j - x_i + y_i - y_j$$
$$x_j - x_i + y_j - y_i$$

When considering Farmer Nhoj's cities j ($n < j \leq 2n$), each city will be covered by one of the 4 cases in comparison to city i. By sweeping from left to right, we can maintain which of Nhoj's cities are in cases 1 and 2, as well as which of Nhoj's cities are in cases 3 and 4. Initially, all cities are case 3 or 4. When we encounter one of Nhoj's cities in the sweep, we switch it to case 1 and 2.

However, we still need to be able to differentiate between case 1 and 2 or case 3 and 4. All of Nhoj's cities with $y_j \leq y_i$ are case 1, and $y_j > y_i$ are case 2. This motivates us to build 4 Segment Trees, one for each case.

For example, for case 1, since we know the distance will be $a_{1,i} + x_i - x_j + y_i - y_j + a_{j,2n}$, we know the minimum distance to a city of case 1 will be $a_{1,i} + x_i + y_i$ (which only depend on i, the city we're fixing)

plus $a_{j,2n} - x_j - y_j$. Thus, for the segment tree of case 1, we can store all the values of $a_{j,2n} - x_j - y_j$, and we will want our segment tree to perform range minimum queries.

Similar logic should follow for cases 2, 3, and 4. We separate the values that depend on i from the values that depend on j, and toss all the j-dependent values onto the segment tree.

When processing city i, we query $(0, y_i]$ for cases 1 and 3. For cases 2 and 4, we query $[y_i, 10^9]$. When we sweep from left to right we must be able to delete cities from cases 3 and 4 and insert them into the segment trees for cases 1 and 2.

There are a few implementation details we must cover related to the segment tree. Since there could be multiple possible cities in one leaf of the segment tree (the cities share the same y_j and are left of the current sweep line), in each leaf of the segment tree we store a multiset to keep track of all the cities on each leaf.

Insertion and deletion are the same as point updates, just we remove or insert on our set in the corresponding leaf (then recalculate the values on each node of course). This only takes $O(n)$ extra memory as we only keep track of $O(n)$ cities. It also only adds an extra $O(\log n)$ factor to updates.

The other issue we must tackle is that the points can range from $[1, 10^9]$. Thus, we must use coordinate compression to make our segment trees take $O(n)$ memory instead of $O(y_i)$ memory. We have now solved the problem in $O(m \log n + n \log n)$, since Dijkstra's algorithm is $O(m \log n)$ and our sweep line will process $O(n)$ cities, and for each city we perform a constant number of segment tree queries and updates that each take $O(\log n)$. Code

14. Permutations - Advanced H

As with all query problems, we first learn to answer the problem without queries. Given an array, how do we calculate the minimum number of operation 2 needed to create a permutation (if the array is length k, it's a permutation in this problem if it contains integers $1 \dots k$ in some order)?

Firstly, we must remove elements from the end of the array while there are duplicates. Otherwise, we wouldn't end up with a permutation. Having done that, we must decide how many more elements we should remove before we use our 2nd operation to finalize the permutation.

Well, if we're given an array and we can add elements to the end to make it a permutation, the smallest number of elements we can add is to add all the missing element smaller than the maximum value in the array. For example, given the array $(2, 5, 3)$, we must add 1 and 4 since those are the only two missing elements less than 5, which is the max.

Thus, if given an array of length k, the minimum number of elements we must add is simply $\max_{1 \le i \le k} a_i - k$. So, after removing all the duplicates from the end of the array, if we calculate this value for each prefix, we want the minimum over all prefixes.

It seems difficult to answer these queries directly, since the maximum on a prefix depends on l. Thus, we are motivated to solve the problem offline, and process all queries sorted by l. We need to be able to maintain $\max_{1 \le i \le k} a_i - k$ as we change l. Thus, it is easiest to iterate from $l = n$ to $l = 1$. If we go backwards, we can maintain a monotonic stack of the suffix maximums. Let's consider what updates happen when we transition from $l + 1$ to l. We can imagine storing the answer for each right endpoint in an array and updating this array.

First, the length of each prefix increases since we have added an element to the beginning of our array. As such, we need to add one less element (for example, say we have array $(4, 5, 3)$ and we add 1 to the front making $(1, 4, 5, 3)$. Now, we only need to add 1 element rather than 2) Thus, we must subtract 1 from the range $(l + 1, n)$, which we can do with a Lazy Segment Tree. The answer at index l with left endpoint at l is of course $a_l - 1$.

Then, we pop all elements on the stack that are smaller than a_l. Let's say the top of the stack is $x > l$. Currently, a_x is the maximum element from index x until index y, if y is the element after x in the stack (if there is no element after x in the stack we can pretend $y = n + 1$). At index $x \le k < y$ the value is currently $\max_{l \le i \le k} a_i - (k - l + 1) = a_x - (k - l + 1)$ since $k - l + 1$ is the length of the subarray (l, k). We want to change this to $a_l - (k - l + 1)$. Thus, we should add $a_l - a_x$ to all elements in the range $(x, y - 1)$. Our lazy segment tree is capable of handling this operation too.

Finally, we can answer all queries with left endpoint at l. If given r, our answer would be the range minimum from (l, r) on our segment tree. However, we must remind ourselves that we must remove all duplicates from the subarray (l, r). Thus, we must also keep track of the first duplicate element in this suffix of l. We can maintain this with a map that stores the most recent index of each element.

As we transition from $l + 1$ to l, we do a constant number of map operations and do a range subtraction. When we pop elements from the stack, we do a range add, and since we pop from the stack an amortized $O(n)$ times, our offline processing takes $O(n \log n)$. Adding in the queries, we solve the problem in $O((n + q) \log n)$.
Code

15. Stones - Advanced J

A common way to solve problems that are of the form "Given an initial state and an a bunch of operations you can do, what other states can you reach (in this case, the state with minimum number of stones)?" is to find an invariant between states and abuse it.

The most obvious invariant would be that operations don't change the number of stones modulo $N-2$. However, the invariant on the sum of stones doesn't explain why a state where all piles are 0 stones except for pile 1 with a stones can't reach the same state with pile 1 having $a+N-2$ stones.

My good friend oursaco had to reveal to me that another invariant. Let a be an array of length N with $a_i = b_i - w_i$. One operation either adds 1 to all elements of a or subtracts 1 from all elements of a. Essentially, the difference array of a is invariant.

This clearly explains why we can't just add $N-2$ stones to a pile through our operations, but are these two invariants enough? We are curious if given two states of piles (b, w) and (b', w') with the same number of stones modulo $N-2$ and the difference arrays a and a' are the same, is (b', w') reachable from (b, w) and vice versa (note that connected states are undirected since we can always undo operations)?

Since we know that we need to transform b into b' and w into w', let's just transform b into b' (clearly possible since we can just remove/add black stones until the match) without worrying about what happens to w for now. We will also ignore the condition that the number of stones in a pile can't go negative. After playing around with some examples, you may realize that when you match b with b', $w_i' - w_i$ is the same for all i

After matching b with b', let $d = w_0' - w_0$. Since $a_1 - a_0 = a_1' - a_0'$, $b_1 - w_1 - (b_0 - w_0) = b_1' - w_1' - (b_0' - w_0')$, we can simplify this equation to

$$b_1 - b_1' - b_0 + b_0' + w_1' - w_1 = w_0' - w_0$$

Since $b_1 = b_1'$ and $b_0 = b_0'$, we have $w_1' - w_1 = w_0' - w_0 = d$. We can then inductively show that $w_i' - w_i = d$.

Furthermore, since we know the number of stones modulo $N-2$ is invariant, we know $Nd \equiv 0 \pmod{N-2}$. If N is odd, then n and $n-2$ are coprime (and thus, $n \bmod (N-2) = 2$ has an inverse modulo $N-2$, and we can find that $d \equiv 0 \pmod{N-2}$. If N is even, then d can be either 0 or $\frac{N-2}{2}$ modulo $N-2$.

It would be nice if we could just add or subtract $N-2$ white stones from each pile. Turns out, we can do that. I was solving this problem together with Keys (yes, I am bad and can't solve any hard problems alone XD) and he already came up with a way to do this.

If we want to add $N-2$ white stones to each pile, we can remove a black stone from all piles except pile i (simultaneously adding $N-2$ white stones to all piles except for pile i, which gets $N-1$ white stones added). We can then remove one white stone from pile i (then adding a black stone to all the other piles), resulting in all piles with $N-2$ extra white stones.

To subtract $N-2$ white stones from each pile, it's not difficult to see the same logic applies, but we invert the operations (when we remove a black stone and add white stones to all other piles, add a black stone and remove a white node from all other piles instead).

Now there are a couple of things we pushed to the side and ignored to make this work. First of all, we haven't addressed that when N is even we could have $d \equiv \frac{N-2}{2} \pmod{N-2}$, and secondly, we ignored the condition that we can't have negative stones.

We may notice that, to add $N-2$ white stones to each pile, if there are at least $N-1$ piles with at least 1 black stone, we can successfully add $n-2$ white stones to each pile without going negative anywhere (so long as there are actually $N-2$ white stones to remove). We are only concerned when there aren't $N-1$ piles with a black stone.

In that case, we remind ourselves that we have an operation that can spawn $N-1$ black stones in separate piles! By removing a white stone from pile i, we can create the $N-1$ black stones that are needed, and then turn those black stones into white stones. Note that we no longer have to rectify pile i (the one pile where we don't turn a black stone into $N-1$ white stones) having an extra stone, as we have returned the white stone we used to generate the required black stones to its orignal location. If that doesn't make sense, try to turn $b = (0,0,0,0), w = (0,1,0,0)$ into $b = (0,0,0,0), w = (2,3,2,2)$ yourself.

We must also consider the case where we don't have enough piles with a black stone, but also have no piles with a white stone. In this case, we must remove one black stone to create $N-1$ white stones. Then, we pick one of those piles i that has a newly added white stone, and create all our black stones by removing a white stone from i. If we remove a black stone from all piles except for i, we will almost have the desired array. We just have to add the black stone we removed originally back into its first pile. Once again, if it's confusing, you can try on the case $b = (1,0,0,0), w = (0,0,0,0)$ becoming $b = (1,0,0,0), w = (2,2,2,2)$.

The one catch is, if there are no stones at all, we can't magically spawn in $N-2$ white stones. But to every other state (assuming that state is valid originally, so no negative stones), we can add $N-2$ white stones to each pile.

When it comes to removing $N-2$ white stones, we aren't that worried about going negative (assuming there are $N-2$ white stones to remove from each pile), except when there is no i such that $w_i > N-2$. In this case, one of the white piles will end up negative in our procedure.

In such a case, as long as we have a black stone, we can turn that black stone into $N-1$ white stones. Assuming that the original state had $N-2$ white stones to remove from each pile, since $N = 3$ there must now exist a valid i such that $w_i > N-2$. We can then remove $N-2$ white stones from each pile and we will be left with an extra $N-1$ white stones, which we can return to the original black pile we took from. Once again, you could consider the case of turning $b = (1,0,0,0), w = (2,2,2,2)$ into $b = (1,0,0,0), w = (0,0,0,0)$ if it doesn't make sense.

Now, we're only concerned when there is no pile i with an extra white stone and we don't have any black stones. The only case where this is possible is with no black stones and each pile has exactly $N-2$ white stones. However, if we were to remove $N-2$ white stones from each pile we would be left with no stones at all, so it makes sense that this state is unreachable.

Thus, we have proven that we can always add $N-2$ white stones to each pile or remove $N-2$ white stones from each pile as long as we don't start or end in a position where there are no stones at all.

We have done a lot of work to prove the validity of adding and removing $N-2$ stones from each pile, but this doesn't really clear our doubts on turning b into b', since some piles of white stones may go negative. However, if we assume that b, w and b', w' are both states with no negative stones (and at least one stone in total), since we can always add $n-2$ white stones to each pile, we may preemptively add a theoretically infinite number of white stones to each pile.

Then, we could match b to b' without any worries, and then remove $N-2$ white stones from each pile as needed until $b, w = b', w'$.

Now we tackle the nasty case of $d \equiv \frac{N-2}{2} \pmod{N-2}$. We wonder if for even N we can add or subtract $\frac{N-2}{2}$ from each pile. It seems like there is some parity factor at work here. The operations seem to equate one white stone to $N-1$ black stones from the other $N-1$ piles.

Let's consider the sum of $w_1 + b_2 + b_3 + \cdots + b_N$. If we add or remove a white stone to pile 1, this sum changes by a $N-2$. If we add or remove a black stone to pile 1, the sum doesn't change. If we add or remove a white stone to a different pile, this sum changes by $N-2$. And lastly, if we add or remove a black stone from a different pile, the sum doesn't change. We conclude that the sum modulo $N-2$ is invariant.

From this, you can see that also $b_1 + w_2 + w_3 + \cdots + w_N$ is invariant modulo $N-2$ as well. In fact, these two conditions together are strictly stronger than the number of stones modulo $N-2$, but I solved the problem before realizing this.

Consider adding or removing $\frac{N-2}{2}$ white stones from each pile. Since only w_1 contributes to the sum, the sum changes by $\frac{N-2}{2}$, which breaks our invariance. Thus, we show that we can only add $N-2$ white stones to each pile, not $\frac{N-2}{2}$.

However, I didn't actually prove this while solving the problem, nor did I come up with the $w_1 + b_2 + b_3 + \cdots + b_N$ until now, so you will see in the solution something a little different.

We must put these invariants together and find b', w' that minimize the number of stones while still being reachable from b, w.

We know that the difference array of a (recall $a_i = b_i - w_i$) will always be the same, and we can calculate that difference array from the input b, w. Then, all possible a that we could reach from b, w are uniquely defined by the difference array and the value of a_0, or $b_0 - w_0$.

From here onwards there are many ways to hack yourself a solution, and I'm not confident that what I have written here is of help to anyone but myself.

Forget the sum modulo $N-2$ for now. How would we minimize the number of stones given the difference array of a? Well, if we knew a, the minimum number of stones would be equal to $\sum a_i$: if $a_i > 0$, we would put a_i black stones in pile i. If $a_i < 0$, we would put a_i white stones in pile a. Furthermore, it's intuitively impossible to have less stones than $\sum a_i$.

So now, given the difference array of a, we would like to find a_0 such that we minimize the sum of a. Imagine $a_0 = 0$ and putting all points of a on a number line. We can simultaneously shift all points to

the left or the right by changing a_0. We can think of it as finding x that minimizes $\sum |a_i + x|$.

If there are more points left of 0 than right of 0, then we can add 1 to all points and it will be more optimal. Similarly, if there are more points to the right of 0 than left of it, we can subtract 1 from all points. If we continue shifting our points until we achieve an optimal answer (equal number of points on both sides of 0), then there exists an optimal solution with the median of the points at 0. Hence, an optimal value for a_0 is the negative of the median of the array $a_i - a_0$.

Let's call this "optimal" value of a_0 x. If $a_0 = x$, we can then find the optimal b, w to minimize the number of stones, which is $Nx + \sum(a_i - a_0)$ (to calculate this I calculate the sum of the black stones separate to the white stones, since if $a_i > 0$ we will have black stones, and white stones otherwise). Now, we can consider the other invariants to see if we can derive a b, w from this one that satisfies our other constraints.

We must first fix the sum modulo $N - 2$ without changing the array a. Note that we can add black stone and one white stone to a pile to increase the sum by 2 but leave a the same. If we do this until the sum is correct modulo $N - 2$, we will have our answer given $a_0 = x$. This answer will be optimal for $a_0 = x$ because we begin with a lower bound of $\sum a_i$, and it's not possible to add an odd number of stones since that would disturb a.

For odd N, there are no other invariants to worry about, and we can always add $y < 2(N - 2)$ such that $y + \sum a_i$ is correct modulo $N - 2$. For even N, you may be reminded about the invariant on $w_1 + b_2 + b_3 + \cdots + b_N$.

As I said before, I hadn't proved that when coding the solution (only when writing the editorial), so for even N, after fixing the sum modulo $N - 2$, I directly checked if matching b' (our answer) to b (the input) would require changing all white piles by $N - 2$ or $\frac{N-2}{2}$. In the case of the former, we would have found our answer to be $y + \sum a_i$ with $y < N - 2$.

I checked this by calculating the change in b and maintaining the sum of w accordingly, then checking if it was correct modulo $N(N - 2)$ (since adding $N - 2$ white stones to all N piles changes the sum by $N(N - 2)$). This required use of 128-bit integers since $(N(N - 2))^2 > 2^{63}$.

If the $w_1 + b_2 + b_3 + \cdots + b_N$ invariant doesn't hold, we must add more stones while maintaining the sum modulo $N - 2$. So best case, can we add $N - 2$ stones to fix it? Indeed, we can. Imagine adding $\frac{N-2}{2}$ white stones to a pile and $\frac{N-2}{2}$ black stones to that same pile. We can trade the $\frac{n-2}{2}$ black stones for $(N - 1)(N - 2)$ white stones (in all the other piles), thus effectively we have added $\frac{N-2}{2}$ to all piles. Thus, $y < 2(N - 2)$ yet again.

Now, we haven't considered picking any other values for a_0 yet. Intuitively, we want to check around a radius of x since straying too far from x would balloon $\sum a_i$.

Recall that our answer depends on $w_1 + b_2 + b_3 + \cdots + b_N$ modulo $N - 2$. We can also determine that $b_1 + w_2 + w_2 + \cdots + w_N$ is invariant modulo $N - 2$. Of course, $w_1 + b_2 + b_3 + \cdots + b_N + b_1 + w_2 + w_2 + \cdots + w_N = \sum a_i$.

In one step, N of these values change (out of the $2N$ values we have), so in 2 steps (in the same direction away from x), at least one of the 2 expressions must have changed by $N - 2$. Notice that for both odd and even N, we fix our invariants by adding 1 black stone and 1 white stone to a pile. This effectively adds 1 to our two separate expressions.

As such, at least one of these expressions is increased by $N-2$ if we move 2 steps. Given $w_1 + b_2 + b_3 + \cdots + b_N$ and $b_1 + w_2 + w_2 + \cdots + w_N$, we are adding 1 to both expressions until they satisfy their original values modulo $N-2$. Thus, if one of them is increased by $N-2$, the new answer can't possibly be better than the old answer.

I don't have a more rigorous argument, but I'll be upfront when I say I'm too stupid and lazy to formulate this intuition into something more convincing (the way I look at it is at $a_0 = x$, the two expressions are like stacks where we need to fill both up until they are correct modulo $N-2$, which is the limit of the stack. If one of them increases by $N-2$, then that stack will have exceeded its original limit, so any new answer can't possibly be better than the old one). Regardless, this means that we only need to check three values for a_0: $x-1, x$, and $x+1$!

Lastly, if we leave our algorithm as is, it may return the answer 0. This is only possible when we initially have no stones, so if our answer is 0 and there are a nonzero amount of stones, our answer is actually $2(N-2)$, which is the next best answer that satisfies the invariants.

We can't achieve $N-2$ because if we add a stone to a pile, then we must add $N-3$ more stones in that same group (either $b_1, w_2, w_3, \ldots, w_N$ or $w_1, b_2, b_3, \ldots, b_N$). Doing this would definitely change the difference array of a, so the next best option is to place $N-2$ black stones and $N-2$ white stones in the same pile.

I used sorting to find the median of the points, and you can precompute some values (or just do it in linear time) to calculate the answer for $a_0 = x-1, x, x+1$. Regardless, we can solve the problem in $O(N \log N)$.

Code

16. Entrance Exam - Advanced K

There are many problems involving counting pairs (l, r) in an array (counting inversions, for example) which are solved through divide and conquer. After trying some ideas that don't seem to work, we're hopefully motivated to try the divide and conquer approach.

We recursively split the array into two halves, and count the number of pairs (l, r) and with $l \leq m$ and $r > m$. We can rewrite the condition in the statement as $a_l + a_r - \min_{l \leq i \leq r} a_i - \max_{l \leq i \leq r} a_i = 0$.

Let's iterate over i such that $l \leq i \leq m$ (it's a bit confusing but now we will let l and r be the boundaries of a in our divide and conquer, and we are counting pairs of (i, j) instead) and for each i we'd like to cound how many valid j there are. The key idea is that we can casework on the max and the min values.

There are 4 cases, based on whether min and max are on the left half or the right half (we will tiebreak by saying if the min or the max are equal in both halves, they go to the left side).

If the left half contains the min value, then all valid j are to the left of the smallest k $(m < k \leq r)$ such that $a_k < a_i$. Thus, j can be $[m + 1, k)$. If the right half contains the min value, then the condition is $j \geq k$, so j can be $[k, r]$. Similar logic applies for the max value, but k $(m < k \leq r)$ is the smallest index such that $a_k > a_i$.

If we iterate from $i = m$ to $i = l$, then the minimum on the left half will be monotonically nonincreasing, and the maximum on the left half will be monotonically nondecreasing. Thus, k_{min} and k_{max} will be monotonically increasing as well. Thus, we can use a two-pointers algorithm to find k_{min} and k_{max} for each i.

For all of the 4 cases, if we know k_{min} and k_{max} for some i, then we want to find all j that satisfy $a_i + a_j - \min - \max = 0$. Let's split up the cases:

For case 1, both the min and the max are on the left side, so we have $a_j = -(a_i - \min_{left} - \max_{left})$. For case 2, the min is on the left side and the max is on the right side, so we have $a_j - \max_{right} = -(a_i - \min_{left})$. For case 3, the min is on the right side and the max is on the left, giving us $a_j - \min_{left} = -(a_i - \min_{right})$. Lastly, for case 4, when both the min and the max are on the right side, $a_j - \max_{left} - \max_{right} = -a_i$.

We can imagine making 4 arrays b_1, b_2, b_3, b_4, where $b_{1,j} = a_j$, $b_{2,j} = a_j - \max_{right}$, $b_{3,j} = a_j - \max_{left}$, and $b_{4,j} = a_j - \max_{left} - \max_{right}$. Then, when we process i, for each case, we need to be able to count the number of occurrences of some number x (for case 1, $x = a_i - \min_{left} - \max_{left}$, and so on) in a range.

We know how to do this offline with a map, since to count the number of occurrences of x from (l, r) in an array is to simply count the range $(0, r)$ and subtract $(0, l - 1)$. Our map would count the occurrence of each x as we iterated from 0 to $n - 1$ (I am being messy with variables but this is just on a general array of length n), and we would easily be able to update our answer offline.

However, this would take $O(n \log^2 n)$, which is too slow. There is an intelligent way to solve this problem in $O(n \log n)$, which you can read about in the official editorial, but for us it suffices to use a hashmap like an unordered map or a gp hash table

I am writing the editorial before culver0412 sets his own time limits for the problem, so I will update this part if I ever need to perform any optimizations.

Edit: Yay, no optimizations needed

By removing the log factor with a hashmap, we solve the problem in $O(n \log n)$.

Code

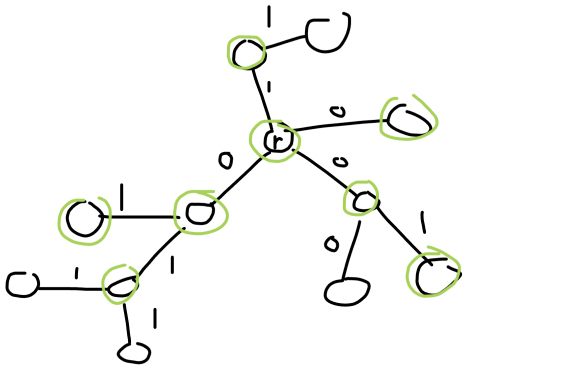

Figure 1: Tree with nodes in S colored green

17. Cool Problem - Advanced L

Please note that I was unable to solve this problem during testing. The problemsetter, culver0412, revealed to me a large component of the solution to assist me. His solution and proof in the official editorial are of a higher quality than mine.

I tried this problem for three days or so before culver0412 gave me the key hint. I genuinely do not know how to motivate this fact, as all the ideas I tried before were along the lines of dynamic programming and some incorrect greedies. However, when it was revealed to me, I wasn't surprised by the statement's verity, as playing around with many, many, examples had led to me to suspect it; I just couldn't concretely prove it.

The main claim is: In the optimal coloring of edges there will always be a node x such that there is a cool path from x to every other node in the tree.

Although his proof is more sophisticated than mine, I think I was able to come up with something after being revealed that nugget of information. A common way to prove that something is optimal is to take something not optimal, and show that you can make it better by doing said "optimal" thing. We will try to use an exchange argument to prove the main claim.

Regardless, I recommend you just try to prove it / convince yourself (or read culver0412's proof) and then solve the problem from there (the rest of the problem is not too difficult, aside from the constant optimization, which was hellish). However, it would not be an editorial without a proof, so here goes.

Let's say we have a tree already colored for us, and we pick an arbitrary node r to be the root (the node which has a cool path to every other node in the tree). Now, there will be a set of nodes S which already have cool paths to r. Of course, since all these nodes are connected to r by a cool path, all the nodes in S are connected.

As an example, in Figure 1, you can see a tree (black edges are 1, white edges are 0, and nodes in S are green) where we have selected some root r.

Now, we want to show that we can slowly include more nodes in S without worsening our answer until all n nodes are in S.

First, we should orient the current nodes in S optimally. In other words, consider all the cool paths which use r as an intermediary node (we assume that intermediary means that the path travels through r but doesn't start or end there). They will start in one node in S and end in a different node (still in S) which happens to be in a different subtree (if we root the tree at r). If there are k nodes adjacent to r (and thus k subtrees of r), let $x_1, x_2, \ldots, x_k$ be the number of nodes in each subtree of r that are also in S.

For example, in the graph above, $x = (1, 1, 2, 3)$, as there are 2 subtrees with 1 node in S, 1 subtree with 2 nodes in S, and 1 subtree with 3 nodes in S. If we let $v_1, v_2, \ldots, v_k$ be the corresponding adjacent nodes to r, still rooting the tree at r, we can flip all the edges in any v_i's subtree (change white to black and black to white) without affecting any cool path that doesn't use r as an intermediary node.

Furthermore, it is optimal to split x into two subsets S_1 and S_2 where you minimize the difference between their sums. This is because the number of cool paths without r as an intermediary node will stay the same, and then the number of cool paths with r as an intermediary node will be the sum of S_1 times the sum of S_2. Let b_r be the sum of subset S_1 and w_r be the sum of subset S_2, and let's assume $b_r < w_r$ without loss of generality.

w_r is essentially the number of cool paths ending at node r where the last edge is white, and b_r is similarly defined where the last edge is black.

If d is the difference between w_r and b_r, then we know that $w_r = b_r + d$ and $w_r + b_r = m$ (where m is the total number of nodes in these subsets), so $b_r + b_r + d = m$. Solving this equation for b_r we get that $b_r = \frac{m-d}{2}$. Thus, $w_r = b_r + d = \frac{m+d}{2}$. The number of cool paths with r as an intermediary node is then:

$$x_1 x_2 = \frac{m-d}{2} \cdot \frac{m+d}{2} = \frac{m^2 - d^2}{4}$$

m is a constant, and the expression $\frac{m^2-d^2}{4}$ is maximized when d is as close to 0 as possible, proving that we want to minimize the difference between b_r and w_r.

Now, having done this (you will see why this is important later), we can start to add nodes to S while not decreasing the total number of cool paths.

Let's consider all nodes u not in S such that u is adjacent to a node in S. If we were to include u in S we would flip the color of the edge from u to its parent as well as all edges in u's subtree.

Note that we don't change the number of cool paths contained which don't have u as an intermediary node. The only edges that change are as follows:

Consider the parent of u, p. Consider all the cool paths ending at p. Without loss of generality, the color of the edge from u to p is white. Then, let w_p be the number of cool paths ending at p where the last edge is white, and let b_p be the number of cool paths ending at p where the last edge is black.

w_p will decrease by some integer y, and b_p will increase by that integer y. y is equal to the number of nodes in u's subtree which will have a cool path to p (and subsequently, r) after flipping the edge from u to p from white to black.

Originally, the number of paths with p as an intermediary was $w_p b_p$. Now, it will be $(w_p - y)(b_p + y) = w_p b_p - y b_p + y w_p - y^2 = w_p b_p + y(w_p - b_p - y)$.

For our answer to not worsen, $w_p \geq b_p + y$ must hold. Now, we know that at least one of the following must be true:

$w_p \geq y + w_r$ and $b_p \leq b_r$ OR $w_p \geq y + b_r$ and $b_p \leq w_r$. This is because p is in S, so if p is included in the w_r total, it gets multiplied by b_r (all of b_r has a cool path to p), and if p is included in the b_r total, it gets multiplied by w_r (all of w_r has a cool path to p).

So, if $w_p \geq \max(w_r, b_r) + y$ and $b_p \leq \min(w_r, b_r)$, then $w_p \geq b_p + y$ will be true, as $\max(w_r, b_r) + y \geq \min(w_r, b_r) + y$. Thus, if p is included in $\max(w_r, b_r)$, we can flip the edge between p and u as well as all edges in u's subtree, increasing the size of S by y.

After flipping all of these edges, we should recalibrate b_r and w_r to minimize d yet again (I will do my best to explain why that is important now, but really, I recommend just reading culver0412's editorial. As I am writing this I am realizing how horrible this looks).

It's possible that we reach some point where are no more u with p included in $\max(b_r, w_r)$ but not all n nodes are in S yet. However, we can still do the above operation, so long as $w_p \geq b_p + y$. Without loss of generality, assume $w_r \geq b_r$ and when we flip u, we flip the edge from u to p from white to black. If we can no longer flip any u whose parent p is included in the total of b_r, we must flip a u whose parent is in included in the total of w_r.

In this case $w_p \geq y + b_r$ is still true. However, our previous bound on b_p was $b_p \leq w_r$. From this, we would not be able to show that $w_p \geq b_p + y$ since $b_r \leq w_r$. However, b_p could be a lot smaller than w_r. In fact, if there is more than one subtree that contributes to w_r, then $b_p \leq b_r$!

Here's why: We know that u and p are in a subtree that contributes to w_r, which is bigger than b_r. If there exists another subtree that contributes to w_r, then $b_p \leq w_r - o$, where o are those other nodes in w_r which are missing. Those o nodes do not have a cool path to p.

This is why we must minimize d. If we assume d is currently as small as possible (and remember that this is optimal), then $w_r - o \leq b_r$. If this were not true, we would not be minimizing d! This is not that hard to prove:

If $w_r - o > b_r$, then the current difference is $w_r - o - b_r$. If we were to add o to w_r, the new difference would be $w_r - b_r$. If we were to add the o to b_r, though, the new difference would be $w_r - b_r - o$, which is guaranteed to be positive and less than $b_r - w_r$, since $w_r - o > b_r$.

Thus, we have shown that if there is another subtree contributing to w_r, then $b_p \leq w_r - 0 \leq b_r$. Then, it's easy to show that $w_p \geq b_p + y$ since $w_p \geq y + b_r \geq y + b_p$.

The only time this operation might fail is when w_r (still assumed to be the bigger subset size) is entirely comprised of one subtree. In this case, we refer back to what we were trying to prove. We want to show that the optimal result has a root r, but that doesn't mean it can be any r.

If w_r is just one subtree, we can shift r to be one node closer to p. First, notice that the tree doesn't change at all. Now, once again, we recalibrate the tree to minimize d. Now, we wonder if it is possible

to add any more u to S. If not, we know once again that w_r (of the new r) is just one subtree, so we move one step closer to some p again (note that it really doesn't matter which p).

Eventually, r will just be p, as we keep moving it closer. Now, notice that u is by default included in S, since u and p (and thus, r) are connected by one edge, which is a cool path. So, we have found a way to include u anyways! We can recalibrate the tree and continue searching for nodes to add.

Though it is not as rigorous as I would like, hopefully you can see how we could work our way towards adding all nodes to S, thus completing our proof.

What we have proved is pretty powerful. If we knew what r was, we would root the tree at r and calculate the number of cool paths that don't have r as an intermediary node. Note that this is the sum of the depths of each node plus 1 (no path can go up and then down the tree except at the root).

Then, we seek to do the recalibration we've already discussed where we partition the subtree sizes (the subtrees of r's adjacent nodes) into two subsets such that we minimize the difference between their sums. If for any c (where c is a child of r), the subtree size of c is greater than or equal to $\frac{n}{2}$, then partitioning the subtrees is trivial, as c will be in its own set and all the other children in the other set.

There's at most one possible root where partitioning is not so easy, the centroid (some trees have 2 centroids, but if a tree has 2 centroids, then one of the subtrees is of size $\frac{n}{2}$). For the centroid, we want to know which split minimizes the difference between subset sizes.

This is an application of the Subset Sum problem, where we want to know all possible sums of a subset given the subtree sizes. We can then pick the one that is closest to an even split. Recall that the sum of our subtree sizes is $O(n)$, so this is a case of the Subset Sum problem with bounded sum. We can solve the Subset Sum problem with a bounded sum in $O(\frac{n\sqrt{n}}{32})$, which, though it may look shaky, is fast enough to solve the problem.

Thus, we can root the tree at 1 initially, and calculate the number of cool paths that don't have r as an intermediary node as well as all the subtree sizes of the tree. Then, we can reroot the tree to test all possible nodes, and if there is a centroid where our trivial solution doesn't work, we run the Subset Sum with bitset algorithm once. Thus, the problem is solved in $O(\frac{n\sqrt{n}}{32})$.

That's actually kind of a lie though. As I said before, this problem's constraints are really nasty, and I required many constant optimizations before getting AC, including, but not limited to:

Using a bitset of size $2 \cdot 10^6 + 1$ instead of $4 \cdot 10^6 + 1$.

Using C++'s basic string class instead of vectors for the adjacency list (I believe culver0412 optimizes the adjacency list by using something called an XOR-linked Tree).

Removing the second DFS required for rerooting by recording the DFS order from the first DFS.

Custom I/O speedups (I used my friend mark's blazingio, which he took from purplesyringa's blazingio). According to culver0412, even the model solution doesn't get AC with standard fast I/O.

Originally I used a DP to calculate the number of paths that don't use r as an intermediary node, which I then realized could be done in just a variable and no DP.

And now (or maybe not, maybe you still need some optimizations or you needed way less, I'm not that good at optimizing but I could be worse), we solve the problem in $O(\frac{n\sqrt{n}}{32})$.

Code