2023 Acton-Boxborough Computing Competition Editorial

1. Let Him Cook

Since Samuel makes as much ice cream as he can, he makes $\lfloor \frac{N}{A} \rfloor$ servings of ice cream. He's left with $N \mod A$ cups of milk, which allows him to make $\lfloor \frac{N \mod A}{B} \rfloor$ cups of chocolate.

2. Feeling Thirsty

To maximize the number of bottles Eric can fill, he should fill the the bottles that take the least amount of time. So we first sort T in non-decreasing order, and then go through and fill as many bottles as we can, ensuring the sum of the times is less than or equal to K.

3. Trash Removal

Notice that we should always use all of the left winds followed by all of the right winds or vice versa. Alternating between left and right winds is not optimal because you can always sweep off more trash by doing all of the same direction winds first. Then, notice that we are essentially always sweeping the trash in a prefix of the cells off the left edge and the trash in a suffix of the cells off the right edge. We can then loop through the "gaps" between trash to try sweeping all the trash off the left edge first and then the right, and vice versa. Also remember to account for the case where the optimal answer is using only one direction of wind to sweep everything off.

4. RGB Lights

If we let red lights be 0, green lights be 1, and blue lights be 2, notice that the operation does not change the sum of all lights under mod 3. Thus, if the sum of all lights under mod 3 is not equal to 0 (all red lights), $N \mod 3$ (all green lights), or $2N \mod 3$ (all blue lights), it is definitely not possible. It then turns out that if the sum all lights is equal to any of them, it is always possible. This can be seen as we can always get all but two lights to equal the same color just by sweeping left to right and toggling each window of three consecutive lights accordingly. Then, by a case analysis, it can be seen that it's always possible to make the last two lights equal to the rest given that the sum of lights condition is satisfied.

5. Hax Xor

Notice that if the operation is applied to the same two indices twice, they both turn into 0s. Thus, we can simply apply the operation twice each to indices 1 and 2, 2 and 3, 3 and 4, all the way to $N-1$ and N. This is a total of $2N-2$ operations, which is guaranteed to be below 10^6 under the problem constraints.

6. Word Game II

Let's build a trie of all legal words. Observe then that the game is equivalent to having the two players walk down the trie one node at a time and alternating turns. The first player to reach a leaf node (or node that marks the end of a word) loses. On this trie, we can then solve the problem recursively. For each node, we want to determine if the player that picks this node wins or loses. We can determine this by looking at all of its children. The current node wins only if all of its children node lose; if any of its children win, the current node loses because the other player can just pick the child node that allows them to win. If the current node does not have any children or marks the end of a word, it loses. Thus, we can do a depth-first-traversal down the tree, each time calculating if the player that picks the current node wins or not. The answer to the problem can then be found from the root node of the trie.

7. Pichu and Factor Tree

First note that for any two numbers a and b that aren't multiples of each other, neither number can be in the subtree of the other. This means that their product, $a \cdot b$, will be a factor of the root node. Then notice that if you have some subset of the numbers $2, 3, \ldots, N$ such that no number in the set is a multiple of another number in the set, the same idea applies, in that the root node is multiple of the product of all numbers in the set.

Note that the numbers $\lfloor \frac{N}{2} \rfloor + 1, \lfloor \frac{N}{2} \rfloor + 2, \ldots, N$ are all not multiples of each other. This tells us that the root node is a multiple the product of all of these numbers, giving us a lower bound. But it turns out that we can always construct a tree with the root node being exactly the product of all of these numbers! For example, if

$N = 10$, the root node value is $6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 = 30240$. Then, we have the children of the root node be 6, 7, 8, 9, 10. Note then that 5 can come from the subtree of 10, 4 from 8, 3 from 6, and 2 from 4. In the general case, all numbers in the lower half can be found from their double. Thus, we can always construct a valid tree of with the root node as the product of $\lfloor \frac{N}{2} \rfloor + 1, \lfloor \frac{N}{2} \rfloor + 2, \ldots, N$.

All that remains is to count the prime factors for $\lfloor \frac{N}{2} \rfloor + 1, \lfloor \frac{N}{2} \rfloor + 2, \ldots, N$ which can be done in $O(N\sqrt{N})$ time or $O(Nlog(N))$ time (the latter using a sieve).

8. Squid Game Ripoff

We will count the contribution of each number. The idea is that each number has a probability of a contestant surviving by being the only one picking it. Let's think about this probability as the "expected" count of survivors on that specific number. If we sum these expected survivors on all numbers that have a chance of being picked, we get our answer.

Now, to calculate this, suppose we have a number that's inside the ranges of K contestants. Let's call the lengths of their ranges as $L_1, L_2, ..., L_K$. The probability that contestant 1 survives by being the sole picker of this number is

$$\frac{1}{L_1} \cdot \frac{L_2 - 1}{L_2} \cdot \frac{L_3 - 1}{L_3} \cdot \ldots \cdot \frac{L_K - 1}{L_k}.$$

If we sum the probabilities for all K contestants, we get the following expression, which gives us the expected count of survivors on this number:

$$\left(\sum_{i=1}^{K} \frac{\prod_{j=1}^{K} (L_j - 1)}{L_i - 1} \right) / \prod_{i=1}^{K} L_i.$$

So all that is left is to compute this quantity for all numbers that can be picked. If we sort the left and right endpoints of each contestants range, we can do a left to right sweep. Notice that between any two consecutive endpoints in the sweep, the expected number of survivors at each number is the same within the range. Let's maintain a couple of calculations while we sweep. Let

$$prod = \prod_{i=1}^{K} L_i,$$

$$sub = \prod_{i=1}^{K} (L_i - 1),$$

$$cur = \sum_{i=1}^{K} \frac{\prod_{j=1}^{K} (L_j - 1)}{L_i - 1}.$$

Each time we reach an endpoint, let x be the distance between this endpoint and the previous endpoint. We add

$$\frac{cur \cdot x}{prod}$$

to our answer. Then, we update $prod, sub, cur$. If the current endpoint is the start of a range with length y, we update as follows:

$$cur = cur \cdot (y - 1) + sub$$

$$prod = prod \cdot y$$

$$sub = sub \cdot (y - 1)$$

Otherwise, if the endpoint is the end of a range with length y, we update as follows:

$$prod = prod/y$$

$$sub = sub/(y - 1)$$

$$cur = (cur - sub)/(y - 1)$$

Note that the following updates maintain the original formulas as new segments are either included or removed. We continue until all endpoints are processed, and then we return the answer.

9. Ping Pong Equilibrium

After an arbitrarily large number of rounds, the configurations will always cycle between some set of configurations. This is because there are a limited number of configurations, so a repeat configuration must be reached after an arbitrarily large number of rounds. Thus, our task is to count the number of configurations that are part of any cycle.

By analyzing some examples, it appears that these cycles always alternate between 2 configurations.

To prove it, we want to show that it is not possible to create a cycle that returns to the same configuration for the first time in more than 2 rounds. Note that player 1 will always be at table 1, and that player 2 will always alternate between table 1 and 2.

Now, it's harder to show that player 3 always alternates between 2 tables. First, the only other possibility is for player 3 to return to the same table after 4 rounds, by losing against 2 other players and then winning against two players. Anything more than 4 rounds is impossible because it would indicate that player 3 needs to lose to more than 2 different players, but only 2 players can beat player 3. So, suppose player 3 is in a cycle that repeats after 4 rounds. Player 3 must then lose to player 1 and 2 on their way down, meaning they must then start at table 1, lose two matches and reach table 3, then win two matches to come back. But in order to lose to both player 1 and player 2 on the way down, player 2 must start at table 3 and win two matches in a row, which indicates that player 2 is not alternating between two tables. This is a contradiction because we know after a large number of rounds, player 2 is always alternating between 2 tables.

Players 4, 5, 6, and on can be shown to alternate between 2 tables using the same reasoning, with the exception of player 2N who is always stuck at table N. Thus, if all players return to their same table after 2 rounds, we have shown that the whole configuration always cycles after 2 rounds.

Now, to count all configurations that cycle after 2 rounds, notice that the loser at table 1 has to be better than the winner at table 3, the loser at table 2 has to be better than the winner at table 4, and so on. We can create a directed graph where edges mean one spot must be higher ranked (have a lower number) than the other. Notice that this graph will always unravel into a start node, end node, and 2 paths of N - 1 nodes each. Player 2N will always be at the start, player 1 at the end, leaving 2N - 2 players left. Out of these players, we need to pick N - 1 players to be on one of the paths. So, the answer is $\binom{2N-2}{N-1}$.

10. Anuj's Goats

For each fence post, calculate the number of enclosing polygons with it as the lowest vertex. To do so, sort all of the points above it using slope in counterclockwise order, breaking ties by putting earlier distances first. We will dp on these ordered points.

Note that the dp essentially builds these convex polygon fragments in the counter-clockwise order. Let dp[i][j] = # of convex polygon fragments with last point j and second last point i, such that the fragment "surrounds" the goats (meaning it does not split any goats). To pick the next point, k, to join in the convex polygon fragment, search later points in the sorted list. A point k works under 2 conditions:

- i -> j -> k is a left turn
- j -> k -> every goat is a left turn

The first condition ensures the polygon is convex. The second condition ensures that the final polygon will enclose all goats. This is because a polygon encloses all of the goats if and only if every side (in counter-clockwise order) forms a counter-clockwise orientation with every goat inside. Thus, for all k under these conditions, dp[j][k] += dp[i][j].

The last thing that is necessary is to ensure that the polygon ends at the starting vertex. To do this, we can simply iterate through the dp states and perform the same checks, making sure i -> j -> starting vertex satisfies the checks.

If implemented directly, this will give a $O(NM^4)$ complexity. However, we can pre-compute for all pairs of fences if it forms a counter-clockwise orientation with all goats in $O(NM^2)$. This then reduces the complexity of the dp to $O(M^4)$.