MITIT 2024 Round 2 Editorial (Advanced Division)

The MITIT Organizers

January 31, 2024

1 Multisets

Problem Idea: Sam Zhang
Problem Preparation: Zixiang Zhou
Analysis by: Sam Zhang

1.1 Subtask 1

Let $C(S, x)$ denote the number of times x appears in multiset S. We present two possible solutions.

1. For each multiset S and value $1 \leq x \leq 10$, store $\min(C(S, x), 10^{15})$. Then operations 1, 2, and 4 can be implemented directly. For operation 3, we perform the operation directly if the affected element appears less than 10^{15} times in that multiset; if the affected element appears 10^{15} or more times, we can just not perform the operation, since it is impossible for the construction of a multiset with only one element to depend on an earlier multiset with 10^{15} or more elements (because each operation can only remove up to 10^9 elements at once, and there are at most $5 \cdot 10^5 < 10^6$ operations).
2. For each multiset S and value $1 \leq x \leq 10$, store the parity of $C(S, x)$. Notice that for all operations, it is possible to determine the parities of $C(S, x)$ just from the stored values of existing multisets. Then to answer queries, we just need to find the unique element appearing an odd number of times.

1.2 Subtask 2

For each multiset S, store the sum of the elements of the multiset modulo $10^9 + 1$. Then the first three operations can be directly implemented, and operation 4 is just printing the sum modulo $10^9 + 1$ which equals the unique element in the multiset.

1.3 Subtask 3

This solution is similar to subtask 2, but we need to use a different binary operation than "sum modulo $10^9 + 1$" that allows treating addition and removal of elements the same way. To do that, we will instead store the **bitwise XOR** of the elements in each multiset.

This solution is similar to the second solution of subtask 1, by storing the parities of each $C(S, x)$ in a "compressed" manner so only one value needs to be stored per multiset.

2 Beavers and Revaebs

Problem Idea: Ray Bai and Danny Mittal
Problem Preparation: Danny Mittal and Claire Zhang
Analysis by: Danny Mittal

Define M to be a bound on r_k for all k. We will use M alongside N to state the runtime complexities of the solutions below.

2.0.1 Subtask 1

For a given assignment of point values to problems, we can check whether it satisfies the constraint that only the N^{th} beaver and N^{th} revaeb have the same score by first computing the scores of each beaver and revaeb by looping through the problems (first forwards, then backwards) and maintaining a running sum of their point values, then adding these scores to a set and checking that the number of elements in the set is $2N - 1$. This takes $O(N)$ time.

Now, for each problem there are at most M choices of point values, meaning that the total number of assignments of point values that satisfy the constraints $l_k \leq p_k \leq r_k$ is at most M^N. We can therefore enumerate all such assignments and check for each one whether they satisfy the second constraint in $O(N)$ time. Since the number of assignments is $O(M^N)$, this takes $O(M^N N)$ time overall.

In this subtask, $N, M \leq 8$, meaning that $M^N N \leq 8^9 \approx 134,000,000$, which is fast enough to pass.

2.1 Idea for Other Subtasks

The central idea for the solutions to the other subtasks is that any valid assignment of point values can be obtained uniquely using the following process.

2.1.1 The process

We build the assignment of point values from both ends, meaning that we maintain a list of point values at the beginning and a list of point values at the end, and at each step of the process we add a point value to the end of one of those lists. Specifically, we always add a point value to the list whose current sum of point values is smaller. In order to obey the constraints of the problem, we make sure that the point value we add is in $[l_k, r_k]$ for the appropriate k and also that the two lists never have the same sum (except at the beginning, when they're both 0).

2.1.2 Example of the process

Here is an example with 5 problems. We start with no point values assigned:

```
. . . . .
```

We then choose to add a point value of 3 to the beginning:

```
3 . . . .
```

Now the beginning has a sum of point values equal to 3, and the end has a sum of point values equal to 0, so we need to add a point value to the end. We choose to add a point value of 5:

```
3 . . . 5
```

Now the beginning has the smaller sum, so we choose to add a point value of 1:

```
3 1 . . 5
```

The beginning is still smaller, so we add 2:

```
3 1 2 . 5
```

The end is smaller now, so we add 2 (note that we cannot add 1, because then both ends would be equal):

```
3 1 2 2 5
```

And that is our assignment of point values to problems.

2.1.3 Executing the process is equivalent to a valid assignment

Clearly, any valid assignment can be attained using this process. It is also almost unique, because at every step of the way, since the beginning and end must have different sums, we are forced to add to the lesser one – except at the beginning, where we have two choices; we can fix this by always adding to the beginning at the first step.

We can further prove that any assignment attained using this process is valid. First, such assignments clearly satisfy the constraints $l_k \leq p_k \leq r_k$ for all k.

Then, suppose that during the process we added b point values to the beginning and so $N - b$ to the end. We know that the scores of beavers solving at most b problems are distinct from the scores of revaebs solving at most $N - b$ problems because we guaranteed it during the process.

Now, let s be the sum of all point values in the assignment. Observe that the score of a beaver solving more than b problems is equal to s minus the score of a revaeb solving less than $N - b$ problems, and that the score of a revaeb solving more than $N - b$ problems is

equal to s minus the score of a beaver solving less than b problems. It follows from this that the scores of beavers solving more than b problems are distinct from the scores of revaebs solving more than $N - b$ problems (except for the beaver and revaeb that solved all the problems, since there are no corresponding rodents that solved 0 problems).

Continuing, suppose that the score of the beaver that solved b problems is more than the score of the revaeb that solved $N - b$ problems (otherwise, just switch beavers with revaebs in the following argument). We can conclude that the scores of beavers solving more than b problems are strictly greater than, and therefore distinct from, the scores of revaebs solving at most $N - b$ problems.

However, because in the process we always add a point value to the side with the currently smaller sum, since we added a b^{th} point value to the beginning, it must be that the score of the beaver solving $b - 1$ problems was smaller than the score of a revaeb solving at most $N - b$ problems. This means that the scores of the revaebs solving more than $N - b$ problems are strictly greater than, and therefore distinct from, the scores of revaebs solving less than b problems.

From these four cases we can see that the scores of beavers and revaebs are all distinct other than the two rodents that solved all the problems. We can thus conclude that the valid assignments are exactly those assignments produced by the process, and therefore we can count the valid assignments by counting the number of ways to execute the process.

2.2 Subtask 2

Define M so that $r_k = M$ for all k. In executing the process, at each step we have M possible values for the point value to add, suggesting that the answer is M^N. However, one of those values might make the two lists' sums equal.

To fix this, we need to observe an important property of the process that will also be used in the later subtasks, which is that because we always add the point value to the list with the smaller sum, and the point value we add is always at most M, even if that list now has a greater sum it can only be greater by at most M. Therefore, at all times the list with the greater sum only has the greater sum by at most M.

Thus, for this subtask since we always select the point value to be one of $1, \ldots, M$, we can note that since the difference between the sums of the two lists is always at most M, and other than the first step the difference is always at least one, the difference must be one of $1, \ldots, M$, meaning that there is always be a point value out of $1, \ldots, M$.

This means that at any step of the process other than the first, we have $M - 1$ choices for the point value. Meanwhile at the first step we still have M choices, so the number of ways to execute the process is $M(M - 1)^N$. We can thus solve this subtask by simply

computing $M(M-1)^N$ modulo 10^9+7, which can be done easily in $O(N)$ time or using modular exponentiation in $O(\lg N)$ time.

2.3 Subtask 3

In this subtask we will directly count the number of ways to execute the process using DP. Our DP state will be (a, b, d) where a is the current amount of point values at the beginning, b is the current amount of point values at the end, and d is the difference between sums of the two lists. We have that $a, b \in [0, N]$, and from the observation in subtask 2 we know that $d \in [-M, M]$, so the number of DP states is $O(N^2M)$.

To transition, we just choose what the next point value is. There are at most M choices, and the choice determines the next state, meaning that for each choice we have a constant time transition. The overall transition from each state is thus $O(M)$, making the overall runtime of the DP $O(N^2M^2)$. For this subtask we have $N^2M^2 \leq 50^2100^2 = 25,000,000$, which is fast enough.

2.4 Subtask 4

The full solution to the problem involves optimizing the DP from subtask 3 by using prefix sums. We transition to a given state (a, b, d) from two types of states: states of the form $(a-1, b, d')$ and states of the form $(a, b-1, d')$.

Thus, when computing the DP values of the states of the form (a, b, d) for a given a, b, we can first compute prefix sums for the states of the form $(a-1, b, d')$ and states of the form $(a, b-1, d')$, then use those sums to compute the value for each state of the form (a, b, d) in constant time.

The computation of prefix sums is $O(M)$, which is also the number of states of the form (a, b, d) for a given a, b. Because of this, the runtime complexity is proportional to the number of states, meaning that it is $O(N^2M)$. Under the constraints in the problems statement, $N^2M \leq 50^22000 = 5,000,000$, which is fast enough.

3 Square Coloring Game

Problem Idea: Sam Zhang
Problem Preparation: Sam Zhang
Analysis by: Sam Zhang

3.1 Subtask 1

A brute-force algorithm works.

3.2 Subtask 2

Call a white square *good* if either it is located at one of the ends of the board, or it is surrounded by two squares of the same color (which must be either red or green). Note that moves can only color good squares, and a white square in the beginning will be colored non-white at the end if and only if it is good. Moreover, since an odd number of good squares are colored on each move, the parity of the total number of moves made is the same as the parity of the number of good squares.

Therefore, Amy wins if the number of good squares is odd, and Aimee wins otherwise.

3.3 Subtask 3

We claim that Amy wins if N is odd and Aimee wins if N is even.

If N is odd: Amy first makes a move that colors the center square of the board (it does not matter if it is red or green). Now whenever Aimee colors a square S, Amy colors the reflection of S over the center square, using the same color Aimee used. By maintaining the mirror symmetry of the board after each of her moves, Aimee ensures that she can always move as long as Amy can, thus winning the game.

If N is even: Aimee uses a similar mirroring strategy after each of Amy's moves (notice the center of the board is not located in a square this time, so unlike in the case with N odd, Amy cannot make a non-mirrorable first move in the beginning).

3.4 Subtask 4

If the board is completely white, then the problem reduces to subtask 3, so assume there is at least one non-white square in the beginning.

Notice that the non-white squares split the board into independent pieces, where each move is made in one piece at a time. Therefore we can compute the nim-values of each piece and combine them using Sprague–Grundy theorem.

There are two kinds of board pieces to consider: ones where the white region is surrounded by two non-white squares (which we call type I), and ones where the white region is surrounded by one non-white square and an endpoint of the board (type II).

We claim that the nim-values are as follows:

- A type I piece with N white squares and surrounded by two equal-colored non-white squares has nim-value 1 if N is odd, and 0 if N is even.
- A type I piece with N white squares and surrounded by two opposite-colored non-white squares has nim-value 0 if N is odd, and 1 if N is even.
- A type II piece with N white squares has nim-value N.

These facts can be proven using induction on N (details omitted).

3.5 Subtask 5

In the full problem, with $K > 0$ possible, it is no longer the case that non-white squares split the board into independent pieces (since a single move can color squares on both sides of a non-white square), so we cannot apply Sprague–Grundy. Instead, we will analyze the game directly.

Lemma 3.1. *Once the (non-white) colors of the first and last squares of the board are determined, the winner of the game is known. More specifically, Amy wins if and only if either (1) the first square has the same non-white color as the last square in the end, and the initial number of white squares is odd, or (2) the first square has different non-white colors as the last square in the end, and the initial number of white squares is even.*

Proof. In the end state of the game, when there are no legal moves, the board must consist of alternating blocks of red and green squares, separated by single white squares. The number of white squares in the end is odd if and only if the first and last blocks have opposite colors.

Now note that Amy has won the game if and only if the number of white squares in the end has opposite parity from the number of white squares at the start, since these are the cases in which an odd number of moves are made, i.e. Amy moved last. □

Moreover, once the second square is colored non-white, the first square must eventually get the same color; the same can be said about the $(N-1)$-th and N-th squares.

We consider three cases:

- At least one of squares 1 and 2 is colored non-white, and at least one of squares $N-1$ and N is colored non-white. Then the colors of squares 1 and N at the end are fixed, and we can determine the winner using the lemma.

- At least one of squares 1 and 2 is colored non-white, but both squares $N-1$ and N are initially white (or the symmetric case with the two ends swapped). Then Amy, on her first move, can color square N either red or green, without coloring any other squares. One of these two colors, by the lemma, will result in Amy winning, so Amy always has a winning strategy in this case.
- Squares 1, 2, $N-1$, and N are all initially white. Then if $N \leq K+3$, N is small enough for the problem to be solved with brute force, so assume $N > K+3$. Note that in this case, it is impossible to fix the colors of both ends of the board in one move, i.e. simultaneously color one of squares 1 and 2 and one of squares $N-1$ and N. Therefore, if either player ever colors one of squares 1, 2, $N-1$, or N, the other player is free to choose the color at the opposite end of the board and thereby win. Therefore it does not make sense for either player to color any of these four squares, so we can delete them and repeat the algorithm.

4 K-Good Subsequences

Problem Idea: Sam Zhang
Problem Preparation: Sam Zhang
Analysis by: Sam Zhang

4.1 Subtask 1

All M-sequences are K-good, so the answer is just L.

4.2 Subtask 2

With $K = 0$, K-good subsequences are constant sequences, so we need to extend a to a sequence that contains each value at most L times. Since there are M possible values, the answer is $M \cdot L$.

4.3 The Key Lemma

Lemma 4.1. *Consider a square grid that is $M + K$ cells wide, with infinitely many rows starting from a bottom row and going upwards. Given the sequence b, for each b_i, in order, consider a rectangle with height 1 and width $K+1$, position it horizontally so there are $b_i - 1$ cells to its left in the grid, and "drop" it from infinity until it hits a rectangle below. Then the length of the longest K-good subsequence of b is the highest index of a row containing a rectangle, where rows are indexed upwards starting from 1.*

Proof. We claim that for each rectangle, the row that the rectangle ends up in is equal to the length of the longest K-good subsequence ending on the element corresponding to the rectangle; this suffices to prove the lemma. Say that our current rectangle corresponds to element b_i; by induction, assume that rectangles for $b_1, \ldots, b_{i-1}$ are already in the positions corresponding to the longest K-good subsequences ending on these elements. Then the rectangle for b_i, when it is dropped, will hit the highest rectangle (corresponding to element b_j for some j) such that $|b_i - b_j| \leq K$. This corresponds to finding the element b_j within K of b_i with the longest K-good subsequence ending on b_j, and adding one to the longest K-good subsequence length, which is precisely how the longest K-good subsequence ending on b_i can be computed. □

With the lemma, the rest of the problem becomes much easier.

4.4 Subtask 3

We need to pack as many rectangles as possible into an initially empty grid. In each row we can pack at most $\left\lfloor \frac{M+K}{K+1} \right\rfloor$ rectangles; since there are L rows that can be used, the answer

is $L \left\lfloor \frac{M+K}{K+1} \right\rfloor$.

4.5 Subtask 4

There is one rectangle in the beginning in the bottom row, with $a_1 - 1$ cells to its left, and $M - a_1$ cells to its right. We can pack $1 + \left\lfloor \frac{a_1-1}{K+1} \right\rfloor + \left\lfloor \frac{M-a_1}{K+1} \right\rfloor$ rectangles into the first row, including the initial one. The remaining $L - 1$ rows can still hold $\left\lfloor \frac{M+K}{K+1} \right\rfloor$ rectangles each, and the answer is

$$1 + \left\lfloor \frac{a_1 - 1}{K+1} \right\rfloor + \left\lfloor \frac{M - a_1}{K+1} \right\rfloor + (L-1) \left\lfloor \frac{M+K}{K+1} \right\rfloor.$$

4.6 Subtask 5

We can directly simulate the rectangle-dropping process, keeping track the highest occupied cell in each column. Then, note that each cell that is below an occupied cell is inaccessible to future rectangles, so we can treat them as occupied as well. Afterwards, for each range of x horizontally consecutive empty cells, we can fit $\left\lfloor \frac{x}{K+1} \right\rfloor$ rectangles in it (by filling these ranges in bottom-to-top order), and the answer is the sum over these ranges, plus the N elements in the beginning.

4.7 Subtask 6

We use some data structures to speed up the subtask 5 solution. First, for each rectangle, we will determine in which row does the rectangle end up. The row number for a rectangle corresponding to an element is just the length of the longest K-good subsequence that ends on that element. Therefore, we can maintain a DP array storing the longest K-good subsequence ending on each value, using a segment tree for range maximum queries. (Note that we must use a segment tree that creates its nodes dynamically, since the DP array can be as long as $M \leq 10^9$ elements.)

Now we perform a line sweep from top to bottom, maintaining the set of intervals of empty cells, using, say, `std::set` in C++. The rows above the topmost rectangle(s) need to be separately treated.

5 Monitoring Beavers

Problem Idea: Zixiang Zhou
Problem Preparation: Zixiang Zhou
Analysis by: Zixiang Zhou

We will discuss the intended full solution. Slower (e.g. $O(NM)$) time implementations are intended to get subtask 1, and implementations that do not find the shortest sequence are intended to get partial points in the subtasks.

The problem can be modeled by a directed graph, where we may flip the direction of any edge as long as all in-degrees are always at least 1.

A lower bound on the number of flips needed is the number of edges that differ in the initial and final configurations. We will describe an algorithm that uses this many operations in most cases (i.e. flips an edge if and only if it must be flipped), then analyze the cases where it fails and describe how to modify the algorithm to handle them.

In the initial configuration, each vertex has at least one edge going into it. Choosing one such edge for every vertex forms a subgraph that is a pseudoforest. The edges in this pseudoforest already satisfy every vertex, so we can reconfigure all other edges into their desired final configuration in an arbitrary order.

Now, consider all vertices v in DFS postorder from the pseudoforests (so each vertex is considered after everything in its subtree). It is always possible to reorient v's parent edge if necessary because if not, then the final configuration did not satisfy v. After this process, the only edges that might not match between the initial and final configurations are the edges in the cycles of the pseudoforest.

Suppose we need to flip an edge $u \to v$ in the cycle. If we cannot perform this flip at this moment, then v has in-degree exactly 1, so there is a $v \to w$ cycle edge that also must be flipped (otherwise, v won't be satisfied in the final configuration). We need to flip $v \to w$ before flipping $u \to v$, but it is possible that we cannot flip $v \to w$ yet because w has in-degree exactly 1. Repeating this reasoning along the cycle, the only case when we get a "deadlock" is when we need to flip *all* the edges in the cycle and it is not possible to flip any cycle edge initially (because all other edges point away from the cycle).

This situation, when there are (possibly several) entire cycles that need to be flipped, is the only situation when this algorithm breaks down and needs to flip more edges. If this happens, let S be the set of vertices that *ever* had in-degree at least 2 throughout all the flips performed so far (note that if a flip $u \to v$ was performed, then both $u \in S$ and $v \in S$). From each cycle, consider the shortest path from any of its cycle vertices (say v_1) to a vertex in S, say $v_1 \to v_2 \to \cdots \to v_k$. If no such path exists, then the component containing this cycle is a pseudotree, so the task is impossible. Otherwise, observe that none of these

shortest paths can intersect because if they do, the first time they intersect is at a vertex with in-degree at least 2, so both paths can be made shorter. Consider rewinding some of the flips we have performed so far until the moment when v_k has in-degree at least 2 (in fact, this can always be either the beginning or the end). We can insert a sequence of flips $v_{k-1} \to v_k$, then $v_{k-2} \to v_{k-1}$, and so on until $v_1 \to v_2$, then the entire cycle, then $v_2 \to v_1$, $v_3 \to v_2$, etc. until we flip $v_k \to v_{k-1}$ back. To find these shortest paths efficiently, we can first run a multisource BFS on the reversed graph starting from S.

We emphasize that it may be more optimal to perform this sequence of flips at the beginning instead of at the end if the closest v_k's in-degree was at least 2 in the initial configuration but became 1 after flipping some edges. The optimality of this algorithm follows because each flip can only move the closest in-degree at least 2 vertex one edge closer to the cycle, and afterwards, these edges must be flipped back.

Remark: The first stage of the algorithm (involving the pseudoforests) can be simplified into just repeatedly greedily flipping any edge that can be flipped. This method can only get stuck when there are "deadlock" cycles that must be flipped, at which point we can move on to the second stage. This may be efficiently simulated with a queue of flippable edges and an array of in-degree counts. In the second stage, we can also BFS directly from each cycle and stop as soon as a node in S is reached, although some care is required to ensure this properly amortizes to $O(N + M)$ when there are many cycles.

Time Complexity: $O(N + M)$.

6 Solving Equations

Problem Idea: Sam Zhang
Problem Preparation: Sam Zhang
Analysis by: Sam Zhang

6.1 Sets 1-7 ($N = 1$)

Since the LHS of the equation is an increasing function of a, we can binary search on a.

6.2 Sets 8-12, 58-60 ($N = 2$, $M \leq 10^7$)

Loop through the possible values of a. Consider one of the equations; with a fixed, the LHS is an increasing function of b. Binary search on b checking if the result works each time.

6.3 Sets 1-40 ($N = K$)

(Idea by Zixiang Zhou) First solve the equations modulo 2 by checking all 2^N choices for the variables. Using this information, solve modulo 4 by noting that each solution modulo 4 has a corresponding modulo 2 solution with each modulo 2 solution generating at most 2^N solutions modulo 4. Repeat this to solve modulo 8, 16, and so on, until the modulus is larger than M. It turns out this can be carried out reasonably efficiently (with the number of solutions modulo each power of 2 not too high) if $N = K$.

There are other possible solutions as well. For example, we may use the fact that if $N = K$, the equations will have few, likely just one, solution in the *real numbers*, allowing us to approximate the solution numerically.

6.4 Sets 1-60 ($N = K$, or $N = K + 1$ and $M \leq 10^7$

First solve the equations modulo small primes p, by checking all p^N possibilities. (The model solution does this for all $p \leq 100$.) Heuristically, this will give approximately p^{N-K} solutions modulo p. Then we select a few primes $p_1, \ldots, p_x$, such that the product $P = p_1 p_2 \cdots p_x$ is greater than M and such that the product of the number of solutions for each prime is not too large. (The model solution does this using some simple heuristics, by preferring primes that are large relative to the number of solutions they have.) Now, using the Chinese Remainder Theorem, we can combine the solutions modulo each p_i into a single list of all solutions modulo P, and we can look for the equations' solutions from this list.

The model solution is able to solve equations 41 through 57 using approximately 10 seconds of computation per set.

6.5 Sets 58-100 ($N = 2$, $K = 1$)

We use a strategy similar to what was used for sets 1-60, but this time, we ask for $P \approx M^{2/3}$ instead of $P > M$. Consider a coordinate plane where one axis represents each variable in the equation. Then the equation can be seen as a decreasing function of one variable with respect to the other.

Divide the plane into square *blocks* of side length $P \approx M^{2/3}$, and divide each block further into square *cells* of side length approximately $\sqrt{P} \approx M^{1/3}$. Since we solved the equation modulo P, for each block, we know the locations of the approximately P solutions modulo P in that block. Furthermore, each cell contains approximately one solution modulo P each, and we can precompute the positions of the solutions modulo P in each cell, as a function of the cell's position relative to its containing block.

Now the equation, as a curve in the plane, traces out a path of $O(M^{2/3})$ cells. We can go through all the cells in the path and check the solutions modulo P in each cell, in approximately $O(M^{2/3})$ equation evaluations.

The model solution takes approximately 2-3 minutes per set to solve test cases like sets 93-100. To speed up the process, it is possible to run multiple instances of the program on the same computer (to utilize the computer's multiple CPU cores), or even have teammates run the program at the same time.

6.6 Solutions for equations with special properties

There are some ad-hoc approaches that only work for certain equation sets and fail for other sets of a similar size that some contestants have likely used, based on their submissions.

- In equations such as those in sets 72, 77, and a few others, the LHS of the equation is a multiple of one of its variables. In this case we may factorize the RHS (depending on the size of the factors, either trial division or Pollard's rho algorithm may be used; alternatively on Unix systems we can use the `factor` command), and guess values for the variable from the list.
- (Likely noticed by team "team...") The equation in set 63 has a special property: it has only one term with maximum degree, and that term ($555b^5$) happens to depend only on a single variable. Since the RHS is, in relative terms, very close to the maximum-degree term, in this case $b \approx \sqrt[5]{\frac{\text{RHS}}{555}}$. It turns out this approximation for b is only 0.11 away from the true value.