

# MITIT 2025-2026 Winter Contest Editorials: Team Round

The MITIT Organizers

December 7, 2025

## 1 M

*Problem Idea: Arul Kolla*

*Problem Preparation: Arul Kolla*

*Analysis By: Albert Wang*

You just need to implement the statement. We can begin by initializing an  $N \times N$  grid of ‘.’s, and set the desired characters to ‘#’:

- Denoting  $(i, j)$  the cell at row  $i$ , column  $j$ , set  $(0, 0), (1, 0), \dots, (N - 1, 0)$  to ‘#’ to draw the left edge,
- $(0, N - 1), (1, N - 1), \dots, (N - 1, N - 1)$  for the right edge,
- $(0, 0), (1, 1), \dots, (\lfloor N/2 \rfloor, \lfloor N/2 \rfloor)$  for the left \ stroke,
- and  $(0, N - 1), (1, N - 1), \dots, (\lfloor N/2 \rfloor, \lfloor N/2 \rfloor)$  for the right / stroke.

This implements the solution in  $O(N^2)$  time.

## 2 P = NP

*Problem Idea: Sam Zhang*

*Problem Preparation: Arul Kolla*

*Analysis By: Albert Wang*

Notice that no matter how we apply operations, the number of occurrences of P in the string does not change. Furthermore, we can never change the number of Ns that occur after the final P.

It turns out that as long as the input strings A, B match in these two invariants, we can always turn one string to the other, e.g. by first deleting all Ns that come before the last P in A and then adding back however many we need.

So, checking the invariants for A, B gives a solution in  $O(N)$ .

### 3 Marbles

*Problem Idea: Arul Kolla*

*Problem Preparation: Eva Sóllilja Einarsdóttir*

*Analysis By: Albert Wang*

The key observation is that a permutation can be decomposed into *cycles*.

Specifically, for any element  $i$ , the sequence  $i, p_i, p_{p_i}, \dots$ , must eventually return to  $i$ , at which point it will become periodic. Furthermore, these sequences (cycles) partition  $\{1, \dots, n\}$ .

So, for each cycle, if it has even length, we can color it alternating red and blue. If it has odd length, we can color all but one vertex alternating red and blue, and use green for the final vertex.

If implemented efficiently, the cycle decomposition can be obtained and colored in  $O(N)$  time.

## 4 Introduction to Number Theory

*Problem Idea: Ditbul Ban*

*Problem Preparation: Ditbul Ban*

*Analysis By: Albert Wang*

Suppose some  $X$  satisfies the property. Let  $D \subseteq A$  the set of all elements at most  $X$  and  $M \subseteq A$  the set of all elements greater than  $X$ . Notice that due to size,  $\text{lcm } D$  must divide  $X$ , because every element of  $D$  divides  $X$ . Likewise,  $X$  must divide  $\text{gcd } M$ , because every element of  $M$  is a multiple of  $X$ . So, after dividing the array into such  $D, M$ , we can pick some valid  $X$  as long as  $\text{lcm } D \mid \text{gcd } M$ .

Therefore, finding  $X$  is equivalent to finding such a partition  $D, M$ . To do so efficiently, we first sort the entries of  $A$ , and compute all the prefix LCMs and suffix GCDs. Then, we can just iterate over all possible cut points and check them all for an  $O(N)$  solution ( $O(N \log N)$  including sorting.)

**5 67**

*Problem Idea: Arul Kolla*

*Problem Preparation: Arul Kolla*

*Analysis By: Arul Kolla*

Suppose we know  $x = a_i a_j$ ,  $y = a_j a_k$ . Then

$$\gcd(x, y) = \gcd(a_i a_j, a_j a_k) = a_j \cdot \gcd(a_i, a_k).$$

Because every pair of different elements is coprime,  $\gcd(a_i, a_k) = 1$ , so  $\gcd(x, y) = a_j$ . Thus if we know the product involving  $a_j$  and its left neighbor, and the product involving  $a_j$  and its right neighbor, we can recover  $a_j$  by taking the greatest common divisor of the two products.

That is, in two queries  $x$  and  $y$ , we can recover three elements  $a_i$ ,  $a_j$ ,  $a_k$ : we use 2 queries for 3 elements.

Thus the number of queries is at most  $Q \leq \lceil \frac{2}{3}N \rceil$ ; for  $N \leq 100$ , this is at most 67.

## 6 Avoid Copyright Infringement

*Problem Idea: Arul Kolla*

*Problem Preparation: Andrii Smutchak*

*Analysis By: Sam Zhang and Andrii Smutchak*

If any of  $X, Y, Z$  are 0, then the task is possible only by alternating between the two available letters. From now on assume  $X, Y, Z > 0$ .

Consider the subsequence  $s$  formed by the M's and T's. If two consecutive elements of  $s$  are equal, there must be exactly I between them, and if they are different, there cannot be I's between them. Moreover, there may be one I at each end of the sequence.

The minimum number of adjacent MT or TM pairs is 1, and the maximum is  $2 \min(X, Z)$  if  $X \neq Z$  or  $2X - 1$  if  $X = Z$ ; all values in between are achievable.

Thus we obtain bounds on the number of I's that we can have (remember to add the two possible I's at each end), and we can check the value of  $y$  against those bounds.

We can construct a subsequence  $s$  satisfying the conditions above by combining blocks of M and T. To construct the final answer we can add I between adjacent equal elements in  $s$ , and in the ends if needed.

## 7 Busy Beaver's Faulty Machine

*Problem Idea: Arul Kolla*

*Problem Preparation: Daniel Kim*

*Analysis By: Daniel Kim*

Note that any base- $B$  integer is congruent to its digit sum modulo  $B - 1$ . This means that if  $Y$  and  $Z$  have the same multiset of digits, they have the same digit sum, which means that  $Y \equiv Z \pmod{B - 1}$ . But  $X + Y = Z$ , which means  $X \equiv 0 \pmod{B - 1}$ .

Now we prove that a solution always exists if  $X \equiv 0 \pmod{B - 1}$ . Consider the following construction, where  $c$  is a large constant (e.g.  $c = N + 5$ ):

$$\begin{aligned} Y &= \frac{X}{B-1} + B^c \\ Z &= B \left( \frac{X}{B-1} \right) + B^c \end{aligned}$$

With this construction,

$$X + Y = X + \left( \frac{X}{B-1} + B^c \right) = B \left( \frac{X}{B-1} \right) + B^c = Z$$

We know that  $Y$  and  $Z$  will have the same multiset of digits because multiplying by  $B$  simply shifts the base- $B$  representation by one position, so  $B \left( \frac{X}{B-1} \right)$  consists of the digits of  $\frac{X}{B-1}$  together with one additional trailing 0. On the other hand, adding  $B^c$  contributes exactly one leading 1 and fills all intermediate positions with 0s, and this contribution is identical in both  $Y$  and  $Z$ . Thus  $Y$  and  $Z$  contain precisely the same collection of digits: both contain the digits of  $\frac{X}{B-1}$  once each, both contain the same leading 1 introduced by  $B^c$ , and both have the rest of their digits filled with 0, so their digit multisets coincide.

## 8 Exam Room

*Problem Idea: Zixiang Zhou*

*Problem Preparation: Samuel Ren*

*Analysis By: Zixiang Zhou*

Subtask 1 can be solved by brute force. Subtask 2 can be solved by an efficiently implemented brute force using bitmasks.

For subtask 3, the key observation is that any valid subset has size at most 5. This observation can be proved using geometric arguments that lead to the full solution. An  $O(N^5)$  algorithm that enumerates all subsets of size at most 5 and checks the condition can pass subtask 3.

For the full solution, sort the points by angle with respect to the origin. We claim that to check if a subset is valid, it suffices to check the condition only for cyclically adjacent pairs  $(P_i, P_j)$  in the ordering. Thus, we can enumerate the first point in the subset and perform dynamic programming, with a state of the last used point. The time complexity is  $O(N^3)$ .

### 8.1 Proof of Geometric Claims

**Lemma 8.1.** *If  $P_i$  and  $P_j$  are both used, then  $\angle P_i O P_j > 60^\circ$ .*

*Proof.* Since  $P_i P_j > OP_i, OP_j$ , by the law of cosines we have

$$\cos(\angle P_i O P_j) = \frac{OP_i^2 + OP_j^2 - P_i P_j^2}{2 \cdot OP_i \cdot OP_j} < \frac{OP_i^2 + OP_j^2 - \max(OP_i^2, OP_j^2)}{2 \cdot OP_i \cdot OP_j}.$$

WLOG  $OP_i \leq OP_j$ . Then, the right hand side becomes

$$\frac{OP_i^2}{2 \cdot OP_i \cdot OP_j} = \frac{OP_i}{2 \cdot OP_j} \leq \frac{1}{2},$$

implying that  $\cos(\angle P_i O P_j) < \frac{1}{2} \implies \angle P_i O P_j > 60^\circ$ .  $\square$

A corollary is that any valid subset has size at most 5, as used in the subtask 3 solution.

**Lemma 8.2.** *Let  $O, A, B, C$  be distinct points. If  $\angle AOC \leq 180^\circ$ , ray  $\overrightarrow{OB}$  is between the acute angle formed by rays  $\overrightarrow{OA}$  and  $\overrightarrow{OC}$ , and  $AB > OA, OB$  and  $BC > OB, OC$ , then  $AC > OA, OC$ .*

*Proof.* We prove that  $AC > OA$ , the other is symmetric. Suppose for contradiction that  $OA \geq AC$ .

If  $OABC$  is a convex quadrilateral (possibly with collinear vertices), then  $OB + AC \geq OA + BC$ . Using the inequalities  $OA \geq AC$ ,  $OB + AC \geq OA + BC$ , and  $BC > OB$  in order gives

$$OB + OA \geq OB + AC \geq OA + BC > OA + OB,$$

a contradiction.

Otherwise,  $B$  lies inside triangle  $AOC$ . Lemma 8.1 implies  $\angle AOC = \angle AOB + \angle BOC > 60^\circ + 60^\circ = 120^\circ$ . Let  $D$  be the foot of the perpendicular from  $O$  to  $AC$ , which lies strictly inside  $AC$ . Either  $B$  lies inside or on the boundary of  $AOD$ , in which case  $AB \leq OA$ , or  $B$  lies inside or on the boundary of  $DOC$ , in which case  $BC \leq OC$ . Both cases contradict our assumptions.  $\square$

The claim follows, since any pair  $(P_i, P_j)$  of used points that is not adjacent in the ordering satisfies  $\angle P_i OP_j \leq 180^\circ$ , and there is a used point  $P_k$  such that ray  $\overrightarrow{OP_k}$  is between the acute angle formed by rays  $\overrightarrow{OP_i}$  and  $\overrightarrow{OP_j}$ .

## 8.2 Solution 2

The  $O(N^5)$  can be optimized to pass using bitset. This is not the intended or recommended solution.

First, notice that the maximum number of subsets possible is about  $(\frac{N}{5})^5$ . This is best seen with sorted angles, with Lemma 8.1 and the maximal arrangement being 5 groups of  $\frac{N}{5}$  points each. This leaves  $(\frac{N}{5})^5$  subsets of size 5, and relatively fewer for smaller subsets.

For each point, precalculate which other points do not interfere, stored in a bitset.

Next, we iterate over triples of points which do not interfere each other. Using the `and` operator returns a bitset of valid fourth points to go to. With the `Find_next` method in bitset can iterate over these points.

Finally, we sum up the popcount of the fourth point's bitset AND the result of the first three. The resulting bitset is a set of valid fifth points, so we just count them up. Since it overcounts by finding previous points as well, we divide this result by 5.

This lets us have a constant factor of  $\frac{1}{5^4 \cdot 64} = \frac{1}{40000}$ . However, this is not quite enough yet.

When we were iterating, the points were in ascending index solutions. This means that for the final set, ideally we only take the suffix of the bitset, which is on average  $\frac{N}{5}$  in size.

However, bitset natively does not support this well enough, so we manually implement the bitset.

An array of 8 64-bit integers works to cover everything, and we use bit operations to AND and iterate across their bits. For the final calculation, we can take the suffix by masking one int and iterating across the rest. This adds another  $\frac{1}{5}$  factor, for a total of  $\frac{1}{200000}$ .

It may also be necessary to manually unroll loops with switch/case.

Using *Ofast*, *unroll-loops*, *fast-math*, *popcnt*, *bmi-2*, *avx2* as pragmas allows this solution to run in about 1 second on Beaver Judge.

## 9 Mahjong Connect

*Problem Idea: Xianbang Wang*

*Problem Preparation: Xianbang Wang and Kangyang Zhou*

*Analysis By: Xianbang Wang*

By discretizing the coordinates, we may assume without loss of generality that all tiles lie in  $[1, N] \times [1, N]$ .

We first consider the special case where there is *only one tile type*. For a tile located at  $(i, j)$ , we interpret it as an *edge* connecting row  $i$  and column  $j$ . This gives a bipartite graph: the left part consists of the  $N$  row vertices, the right part consists of the  $N$  column vertices, and each tile corresponds to one edge between its row and its column.

Under this interpretation, two tiles can be paired *if and only if* their corresponding edges share a common endpoint (i.e., they lie in the same row or the same column). Therefore, the original problem reduces to the following purely graph-theoretic problem:

Given a simple graph  $G$ , determine whether all edges can be partitioned into pairs such that the two edges in each pair share a common vertex. If possible, construct such a pairing.

**Claim 9.1.** *Such a pairing exists if and only if every connected component of  $G$  contains an even number of edges.*

*Proof.* The necessity is immediate: each pair consumes exactly two edges, so every connected component must contain an even number of edges.

We now prove sufficiency. Since each connected component can be handled independently, we may assume that  $G$  is connected.

**A Simple Case:  $G$  is a tree.** Root the tree at an arbitrary vertex. Consider a vertex  $u$  of maximum depth; then  $u$  is a leaf. Let its parent be  $f$ .

- If  $f$  has another child  $u'$ , then  $u'$  is also a leaf. We simply pair the two edges  $(f, u)$  and  $(f, u')$ . Removing these two edges and the two leaf vertices still leaves a tree.
- If  $f$  has only one child  $u$ , then since the total number of edges is even,  $f$  cannot be the root. Let  $f'$  be the parent of  $f$ . We pair the edges  $(f, u)$  and  $(f, f')$ , and then remove these two edges and vertices  $u, f$ . The remaining graph is still a tree.

In both cases, we remove exactly two edges while preserving the tree structure. Repeating this process, all edges can eventually be paired.

**The General Case.** For a general connected graph  $G$ , we first fix an arbitrary spanning tree  $T$ . Every edge of  $G$  is either a tree edge or a non-tree edge. For each non-tree edge  $(u, v)$ , we arbitrarily *assign* it to one of its endpoints, say  $u$ , and conceptually treat it as a “leaf edge” attached to  $u$ . By doing this to all non-tree edges, we yield a tree  $T$  from  $G$ , reducing to the simple case above.  $\square$

Now perform a bottom-up DFS on this augmented tree. For each vertex  $u$ , after all its children have been processed, we consider:

- All unmatched leaf edges attached to  $u$ ,
- The tree edge between  $u$  and its parent (if it exists).

All these edges are incident to  $u$ , so we can greedily pair them at  $u$ . If the total number is even, all are paired locally. If it is odd, we leave exactly one unpaired and pass it upward to the parent. Since the total number of edges in the component is even, this process guarantees that the root will eventually have zero unpaired edges. Hence all edges are successfully paired.

Now we return to the full problem where multiple tile types exist. We again construct a bipartite graph, but now the vertices no longer represent entire rows and columns, but rather *sub-rows* and *sub-columns*.

A *sub-row* is defined as a maximal contiguous segment in a fixed row consisting entirely of tiles of the same type. More precisely, a segment from  $(x, l)$  to  $(x, r)$  is a sub-row if all tiles in that interval have the same type, and the segment cannot be extended further left or right without encountering a different type. A *sub-column* is defined analogously.

Each tile belongs to exactly one maximal sub-row and exactly one maximal sub-column, so we connect these two vertices by an edge corresponding to the tile.

The total number of maximal sub-rows and sub-columns is  $O(N)$ , since each row and column contributes at most as many segments as it has tiles; two tiles are pairable in the original problem if and only if their corresponding edges in this graph share a common endpoint (i.e., they lie in the same sub-row or the same sub-column).

Thus, for each tile type independently, we apply Claim 9.1 to its corresponding bipartite graph. If every connected component has an even number of edges, a perfect pairing exists for that type; otherwise, no valid solution exists.

To conclude, we analyze the time complexity. We identify all maximal sub-rows and sub-columns by sorting the tiles by  $x$ -coordinate and  $y$ -coordinate, respectively. This takes  $O(N \log N)$ . The pairing procedure itself is linear. Therefore, the overall time complexity is  $O(N \log N)$ .

## 10 Collatz Conjecture

*Problem Idea: Albert Wang*

*Problem Preparation: Benson Zhan Li Lin*

*Analysis By: Albert Wang*

We first write down the Collatz sequence started on each of the  $N$  elements, and note the parity of its terms. Let  $a_{ij}$  to be the parity of the  $j$ -th term starting on  $a_i$ ; denote odd parity with  $-1$  and even parity with  $+1$ . Our game is now equivalent to the following process:

- At step  $t$ , we need to take some next  $a_{ij}$  of the correct sign where  $a_{i1}, \dots, a_{i(j-1)}$  have already been chosen.

It is not immediately clear that this process terminates; however, we will show that this is the case. Begin by noting that in any Collatz sequence, every odd number is followed by an even number, because  $3x + 1$  has the opposite parity as  $x$ . Thus all  $-1$ s are followed by  $+1$ s in the sequences of  $a_{ij}$ .

**Claim 10.1.** *Suppose that the game terminates. Then, the optimal sequence of moves must have even length.*

*Proof.* We proceed by contradiction. If we made an odd number of moves, our last move was on some odd number  $x$ . However, that number has now been replaced with  $3x + 1$ , which is even, and hence an additional move may be made.  $\square$

So, we further restrict our attention to only even-length sequences of moves.

**Claim 10.2.** *Suppose we wish to take a prefix of the first  $b_i$  of each sequence  $a_{i1}, a_{i2}, \dots$ . We claim that we can take exactly this set of elements if and only if*

- At least one of the  $a_{ib_i}$  (the last taken element of each sequence) is even,
- The sum of the taken  $a_{ij}$  is zero.

*Proof.* We begin by showing that the two conditions are necessary. First, as our moves alternate in parity, we immediately get the second condition. As for the first condition, our last move must have been one of the  $a_{ib_i}$ s, and therefore at least one of them must have been even.

To show that the two conditions are sufficient, we can simply induct down with casework, by identifying some last two moves that maintain the inductive hypothesis. For brevity, this section of the proof has been left as an exercise.  $\square$

**Corollary 10.3.** *Within the limits of the problem ( $N \leq 500, a_i \leq 10^6$ ):*

- *The sum of any prefix is at least  $-1$ .*
- *We can never take a prefix that sums to at least  $500$ . In particular, we can never select prefixes of elements of more than length  $2026$ .*

*Proof.* The first point can be shown by noting that all  $-1$ s are followed by a  $+1$ .

The first part of the second point then follows immediately, since the total sum of the at most  $500$  prefixes would be positive, and thus nonzero. To check the second part, we can verify that the Collatz sequence started on a number at most a million stabilizes after  $524$  moves to the sequence  $1, 2, 4, 1, \dots$ . Therefore, a prefix of length  $524 + x$  must have sum at least  $-1 + \lfloor x/3 \rfloor$ , and hence  $x \leq 1502$ .  $\square$

Aside from implying that the game must terminate, we can now use a knapsack DP to maximize the sum of the lengths of prefixes that we take, subject to the constraints of 10.2. In particular, 10.3 implies that no partial solution may have a sum outside the bounds  $[-N, +N]$ . So, we can do a DP with two states

- the current prefix  $1, \dots, i$  of the  $a_i$ ,
- the current sum

that stores longest sums of prefix lengths. Accounting for expanding out the first  $2024$  terms of each sequence, we get a runtime of  $O(N^3 + 2026N) = O(N^3)$ .

## 11 Busy Beaver's Dam Logs

*Problem Idea: Anton Trygub*

*Problem Preparation: M V Adithya*

*Analysis By: Yifan Kang*

**Claim 11.1.** *Let  $m$  and  $M$  be the minimum and maximum subarray sum with the same parity as  $x$ . It is possible to find a subarray with sum exactly  $x$  if and only if  $m \leq x \leq M$ .*

*Proof.* Only if direction is trivial by definition. Let  $S(l, r)$  denote the subarray sum of interval  $[l, r]$ :

$$S(l, r) = \sum_{i=l}^r a_i$$

Let  $[L_1, R_1]$  be the subarray with sum  $m$  and  $[L_2, R_2]$  be the subarray with sum  $M$ . If  $x = m$  or  $x = M$ , we are done. Thus we consider the case where  $m < x < M$ .

Let  $\delta_l$  be the sign of  $L_2 - L_1$  and  $\delta_r$  be the sign of  $R_2 - R_1$ . We will operate on the two intervals while maintaining the parity of their respective subarray sum, and the relation that

$$S(L_1, R_1) < x < S(L_2, R_2)$$

Let

$$\Delta = |L_2 - L_1| + |R_2 - R_1|$$

If we can do this while decreasing  $\Delta$ , the operations will eventually terminate at  $S(L_1, R_1) = x = S(L_2, R_2)$ . First, let us consider the case where  $L_1 \neq L_2$  and  $R_1 \neq R_2$ . WLOG  $L_1 < L_2$  and  $R_1 < R_2$ , since other cases can be proven similarly. Consider the operation:

- If  $a_{L_1}$  is even, increase  $L_1$  by 1.
- Otherwise, if  $a_{R_1+1}$  is even, increase  $R_1$  by 1.
- If both  $a_{L_1}$  and  $a_{R_1+1}$  is odd, do both steps.
- If  $a_{L_2-1}$  is even, decrease  $L_2$  by 1.
- Otherwise, if  $a_{R_2}$  is even, decrease  $R_2$  by 1.
- If both  $a_{L_2}$  and  $a_{R_2}$  is odd, do both steps.

It is clear that after one step, we will still have the same parity, and  $\Delta$  has decreased because the indices are closer. Moreover, the value of subarray sums differ by at most 2. Thus, we still satisfy the condition that

$$S(L_1, R_1) \leq x \leq S(L_2, R_2)$$

Notice that if any of them are equal, we are done. Thus, we may still assume that strict inequality holds.

While the above operation works when  $L_1 \neq L_2$  and  $R_1 \neq R_2$ . We need to modify our operations a bit when one endpoint meets. Let us consider the case where either  $L_1 = L_2$  or  $R_1 = R_2$ . By symmetry, WLOG  $L_1 = L_2 = L$  and  $R_1 < R_2$ . Consider the operation:

- If  $a_L$  is even, increase  $L$  by 1.
- Otherwise if  $a_{R_1+1}$  is even, increase  $R_1$  by 1.
- Otherwise if  $a_{R_2}$  is even, decrease  $R_2$  by 1.
- If all are odd, then increase  $L$  by 1, increase  $R_1$  by 1, and decrease  $R_2$  by 1.

It is clear that the parity of subarray sums are maintained, and the  $\Delta$  decrease in one operation.

Finally, we can claim that this process eventually stops, and we have proven that there exists a solution for  $x$ . □

Thus, we can implement a segment tree that maintains the minimum and maximum subarray sum that is even and odd. This allows us to check whether there exists a construction for  $x$  in  $O(N \log N)$ . To find the construction, one can first perform a binary search to get the minimum index  $R$  such that there exists an answer in  $[1, R]$ . Then they can binary search on  $L$  to get the maximum index that there exists an answer in  $[L, R]$ .

## 12 Snacks Scheduling

*Problem Idea: Anton Trygub*

*Problem Preparation: Xianbang Wang, Chunji Wang, and Kangyang Zhou*

*Analysis By: Chunji Wang and Xianbang Wang*

We first introduce a formal version of the problem:

Given an array  $A_1, A_2, \dots, A_N$ , find the smallest possible number of inversions in a permutation  $(P_1, P_2, \dots, P_N)$  such that  $P_i \neq A_i$  for all  $1 \leq i \leq N$ , or determine that no such permutation exists.

**Lemma 12.1.** *For any permutation  $(P_1, \dots, P_N)$ , we have*

$$\#\text{inversions} \geq \frac{1}{2} \sum_{i=1}^N |P_i - i|.$$

*Proof.* Consider transforming the identity permutation  $(1, 2, \dots, N)$  into  $P$  using adjacent swaps. Each adjacent swap that exchanges  $P_j < P_{j+1}$  to  $P_{j+1}, P_j$  increases the inversion count by exactly 1.

Now observe the effect of such a swap on the sum  $S = \sum_{i=1}^N |P_i - i|$ . Only positions  $j$  and  $j+1$  change, and each  $|P_i - i|$  can change by at most 1, so  $S$  changes by at most 2 at each swap.

Let  $K$  be the number of adjacent swaps needed to obtain  $P$  from the identity. Then, using the analysis above, we have  $\text{inversions}(P) = K$ ,  $S \leq 2K$ , proving the Lemma.  $\square$

**Lemma 12.2.** *For any permutation  $P$ , consider a directed graph with  $N$  nodes, and edges  $i \rightarrow P_i$  for all  $1 \leq i \leq N$ . This graph decomposes into cycles  $\mathcal{C}_1, \dots, \mathcal{C}_t$ . For each cycle  $\mathcal{C}_j$ , let*

$$M_j = \max \mathcal{C}_j, \quad m_j = \min \mathcal{C}_j.$$

*Then*

$$\#\text{inversions} \geq \sum_{j=1}^t (M_j - m_j).$$

*Proof sketch.* Consider a single cycle  $\mathcal{C}_j$ . Using the structure of the cycle and the fact that every element in the cycle must move from its original index to its target index, one can show

$$M_j - m_j \leq \frac{1}{2} \sum_{i \in \mathcal{C}_j} |P_i - i|.$$

Summing this inequality over all cycles and applying Lemma 12.1 gives the proof.  $\square$

**Lemma 12.3.** *If  $A_1, \dots, A_N$  is not a constant array, then there exists a permutation  $P_1, \dots, P_N$  with  $P_i \neq A_i$  for all  $i$  and with at most  $N - 1$  inversions.*

*Proof.* We outline an inductive construction. Let  $0 \leq k \leq N - 1$  be the largest integer such that  $A_1 = A_2 = \dots = A_k = 1$ . We try to construct  $P$  so that

$$P_{k+1} = 1, \quad P_i = i + 1 \quad \text{for } 1 \leq i \leq k.$$

This guarantees that the values  $1, 2, \dots, k+1$  appear in positions  $1, \dots, k+1$  but are shifted by at most one position. If the suffix  $A_{k+2}, \dots, A_N$  is not constant, we can recursively apply the same reasoning to this subarray and construct the remaining part of the permutation.

A symmetric argument can be made from the right side: consider the largest index  $l$  such that  $A_l = A_{l+1} = \dots = A_N = N$ , and shift from the end if needed.

If these reductions from both sides are not possible, then the array  $A$  must have the special form

$$1, 1, \dots, 1, x, N, N, \dots, N$$

for some value  $x$ . Without loss of generality, assume  $x \neq 1$  and that there is at least one 1 at the front. Then we can explicitly construct

$$P = (2, 3, \dots, k, N, 1, k+1, k+2, \dots, N-1),$$

where  $k$  is the number of 1's in front of  $x$ . One checks that  $P_i \neq A_i$  for all  $i$  and  $\text{inv}(P) = N - 1$ .  $\square$

From Lemma 12.3, if  $A$  is not constant, the answer is at most  $N - 1$ . If  $A$  is constant, there is clearly no solution (because the single value must appear somewhere, and at that position it would equal  $A_i$ ).

Using Lemma 12.2, if the final answer is  $N - m$ , then the permutation can be decomposed into exactly  $m$  intervals

$$[l_1 = 1, r_1], [l_2 = r_1 + 1, r_2], \dots, [l_m = r_{m-1} + 1, r_m = N],$$

such that for each interval  $[l_i, r_i]$ ,  $\{P_{l_i}, \dots, P_{r_i}\} = \{l_i, \dots, r_i\}$ , and the number of inversion pairs contributed by this interval is exactly  $r_i - l_i$ .

Intuitively, each interval behaves like a “local block” that is internally as inverted as a minimal structure allows, and the total inversion count is  $\sum_{i=1}^m (r_i - l_i) = (N - m)$ .

By Lemma 12.3, such an interval  $[l_i, r_i]$  can be realized (i.e., we can find a valid local permutation within it) if and only if  $A_{l_i}, \dots, A_{r_i}$  are not all equal to a single value  $x$  with  $l_i \leq x \leq r_i$ .

Now think of writing  $1, 2, \dots, N$  from left to right, and placing *borderlines* to separate these intervals. The borderline between position  $i$  and  $i + 1$  is denoted by  $\ell_i$ .

Our goal is as follows.

Place as many borderlines  $\ell_i$  as possible, while avoiding the existence of any interval  $[l, r]$  such that

$$A_l = A_{l+1} = \dots = A_r = x, \quad \text{with } l \leq x \leq r,$$

that becomes an entire interval in the partition.

We introduce the notion of a *block*:

$$[l, r] \text{ is a block if } A_l = \dots = A_r = x, \quad A_{l-1} \neq x, \quad A_{r+1} \neq x, \quad \text{and } l \leq x \leq r.$$

For such a block, it is forbidden to place borderlines both “to the left” and “to the right” of  $x$  inside the block in such a way that  $x$  ends up strictly inside an interval where all values are equal to  $x$ . Concretely, for each block  $[l, r]$  with value  $x$ :

- Either all borderlines in  $\{\ell_{l-1}, \dots, \ell_{x-1}\}$  must be removed,
- Or all borderlines in  $\{\ell_x, \dots, \ell_r\}$  must be removed.

We want to choose, for each block, which side to “sacrifice” (i.e., from which side we remove all borderlines), in order to minimize the *total* number of removed borderlines. Since the total number of potential borderlines is  $N - 1$ , maximizing the number of kept borderlines is equivalent to minimizing the removed ones.

Note that blocks are not fully independent: if two blocks are adjacent, for example  $[u, v]$  and  $[v + 1, w]$ , they share the borderline  $\ell_v$ , so our decisions for these two blocks must be consistent. This interaction suggests a dynamic programming solution. Process the blocks from left to right. For each block, we have two choices:

- Remove all borderlines on the left side of its special position  $x$ , or
- Remove all borderlines on the right side of  $x$ .

It is clear that we can use DP[block index][0/1] to solve this problem. Note an important implementation detail: if  $l = 1$ , we MUST remove right borderlines; if  $r = N$ , we MUST remove the left.

Finally, if the minimum number of removed borderlines is  $R$ , then the answer to the original problem is  $R$ . We can find all blocks in  $O(N)$  time by scanning the array once and grouping maximal constant segments that contain their value index  $x$ . The DP over blocks also runs in  $O(N)$ . Therefore, the total time complexity is  $O(N)$ .

## 13 RMQ

*Problem Idea: Zixiang Zhou and Kangyang Zhou*

*Problem Preparation: Zixiang Zhou*

*Analysis By: Kangyang Zhou and Zixiang Zhou*

Asymptotically, for very large  $N$ , the optimal fraction is actually  $1 - \frac{1}{2e} \approx 0.816$ . Many different approaches that get various partial points are possible. We describe only the full solution (due to Kangyang Zhou) below.

Consider building the Cartesian tree for the array. The simplest idea is to find the minimum of the array, and then split the array into two pieces. Using binary search, it takes  $\frac{\log(\text{len})}{\text{len}}$  cost to find the index of the minimum, which is not good enough. We are willing to accept a minimum value near the middle of the array, so we can set a value  $B$  (which can be  $\sim \sqrt{N}$ ) so that we are “happy” to accept the minimum value of the array that lies in  $[B + 1, \text{len} - B]$ . More specifically, we first find the minimum value ( $m$ ) & place ( $p$ ) among  $[B + 1, \text{len} - B]$ . Then, we deal with the values that lie in  $[1, B]$  and  $[\text{len} - B + 1, N]$ . Take  $[1, B]$  as an example, we enumerate  $x$  from 1 to  $B$ , calculating the minimum value of  $[x, \text{len} - B]$  one by one, until the minimum value equals  $m$ . Assuming that the  $x$  on the left side is  $l$  and the  $x$  on the right side is  $r$ , we will obtain two new intervals that we need to address later:  $[l, p - 1]$  and  $[p + 1, r]$ . Thus, when there are many intervals, we take the one with the largest length to operate on each time.

### 13.1 Correctness Proof

In this problem, we waste at most  $NB$  intervals due to ignoring minimums in  $[1, B]$ , and we waste  $\sum(r_i - l_i)^2/2$  due to the lack of cost. The first term is small enough for us to ignore, so we can simply consider the second term.

We can assume that the intervals we have operated on has  $O(N)$  length, because we have  $1 - o(1)$  waste otherwise. Each time we split an interval,  $\sum \max(r_i - l_i - B)$  will be decreased by at least  $B$ , so the number of splits is  $\leq \frac{N}{B}$ , and the total cost from binary search will be less than  $O(\frac{N}{B} \cdot \frac{\log N}{N})$ , which is small enough for us to ignore. Thus, the only important cost contribution comes from the numbers on the boundary ( $[1, B]$  and  $[\text{len} - B + 1, \text{len}]$ ). Because  $B \ll \text{len}$ , we can treat the operation as taking  $\frac{1}{\text{len}}$  cost to address one number on the boundary.

Thus, we can reorganize the cost calculation into the following: Maintain some intervals. Each time, we take the largest interval, and we can split it into two (with no cost), or use  $\frac{1}{\text{len}}$  cost to decrease the length of an interval by 1. The process stops when the total cost is bigger than 1. We need to estimate the maximum possible value of  $\sum(r_i - l_i)^2/2$  in the final state.

We can see that doing the “split” operation first is favored, because it takes higher cost to

decrease the length of smaller interval. Thus, we can reformulate the problem again: we have some values  $\sum a_i = N$  ( $a_i$  are positive real numbers for convenience), we will take the largest number  $x$  each time and spend  $\epsilon/x$  cost to decrease it by  $\epsilon$  ( $\epsilon$  is a small value that tends to zero), and the process stops when the total cost exceeds 1. We want to maximize  $\sum a_i^2/2$  in the final state.

Until now, we have proved that the solution is optimal, because we can tell that there is something very small at the beginning, and our algorithms achieve the optimal bound for such cases. Therefore, the only thing we need to do is calculate what the answer is.

In the best solution, the operated numbers should have the same initial value; and there should be at most one non-operated value. To calculate the maximum possible value, we enumerate the number  $c$  of changed variables. We find the maximum possible  $x^2 + (ye^{-1/c})^2$  such that  $x \leq ye^{-1/c}$  and  $x + cy = 1$ .

By enumerating  $c$  (the best  $c$  should be a small nonnegative integer), we find that the worst case is  $c = 2$ ,  $x = 0$  and  $y = 1/2$ , which has the value  $1 - \frac{1}{2e} \approx 0.816$ . This corresponds to two separate intervals, each with length  $N/2$ .

## 13.2 Theoretical Upper Bound

A fraction of  $1 - \frac{1}{2e}$  is theoretically optimal, and a direct proof of this is as follows: Consider arrays where the minimum is at  $N/2$ , and the subarrays to the left and right of this minimum are bitonic (first increasing, then decreasing). Even if we are told the entire relative ordering/Cartesian tree of the array, after spending  $\frac{1}{N}$  cost to find the minimum value (giving  $N^2/4$  intervals), it takes  $\frac{1}{N/2-k}$  cost to recover  $N/2 - k$  intervals for  $k = 1, 1, 2, 2, 3, 3, \dots$ . We cannot recover more than  $1 - \frac{1}{2e}$  fraction of the range minimums with cost  $\leq 1$ .