

- Main Contest Link: <https://toph.co/c/mtb-presents-aust-inter-university-2025>
- Practice Link: <https://codeforces.com/gym/105723>
- Onsite Teams: https://therealbcs.com/team-ratings?contest_id=aust_iupc_2025

The contest was intentionally slightly easier than usual. We hope that you liked the problems!

The contest was coordinated by Shahjalal Shohag and reviewed by Jubayer Nirjhor. Problems were reviewed and selected by both of them.

The following is the list of all authors, developers, and testers.

Problem	Author	Developer	Tester
A. Max-Min Madness	Jinlong Han	Jinlong Han	Shaikat Hosen Rony
B. The Absolute MEX Challenge	Shahjalal Shohag	Al-Shahriar Tonmoy	Kawsar Hossain
C. Palindromic Palindrome Partition	Shahjalal Shohag	Mahdi Hasnat Siyam	Al-Shahriar Tonmoy
D. Strong Tree	Ayon Shahrier	Ayon Shahrier	Sabbir Rahman Abir
E. Aloy and the Forbidden Code	Shahjalal Shohag	Kawsar Hossain	Rudro Debnath
F. Rotating Painter	Tamim Ehsan	Tamim Ehsan	Kawsar Hossain
G. GCD and LCM in Perfect Sync	Jinlong Han, Shahjalal Shohag	Jinlong Han	Tasmeem Reza
H. Flip to Zero	Jubayer Nirjhor, Shahjalal Shohag, Al-Shahriar Tonmoy	Al-Shahriar Tonmoy	Mahdi Hasnat Siyam
I. The Art of Rearrangement	Shahjalal Shohag	Rudro Debnath	Al-Shahriar Tonmoy
J. No Duplicates	Sabbir Rahman Abir	Sabbir Rahman Abir	Ayon Shahrier
K. Primal Brackets	Shaikat Hosen Rony	Shaikat Hosen Rony	Rudro Debnath

We also tried to provide problems with multiple variations and with a balanced difficulty. But problem K turned out to be way harder than we expected.

The following is the list of the same problems but in sorted order of (expected) difficulty.

Problem	Expected Difficulty	Expected #AC	#AC (Main Contest)	Tags
E. Aloy and the Forbidden Code	Div3A	130	130	Giveaway
I. The Art of Rearrangement	Div2A	120	127	Adhoc
B. The Absolute MEX Challenge	Div2B	80	114	Constructive
K. Primal Brackets	Div2C	50	10	Adhoc, DP on Trees
F. Rotating Painter	Div2C	40	42	Geometry
G. GCD and LCM in Perfect Sync	Div2D	30	19	Number Theory, Inclusion Exclusion
H. Flip to Zero	Div2D	20	6	DP, BFS, Data Structures
C. Palindromic Palindrome Partition	Div2D	15	5	DP, Strings
D. Strong Tree	Div2E	10	4	Trees, DSU, Segment Tree
A. Max-Min Madness	Div2E	5	6	Adhoc, Combinatorics
J. No Duplicates	Div2E	5	0	Counting, Data Structures, Implementation

Solutions to all problems can be found here:

https://github.com/ShahjalalShohag/bcs-contest-resources/tree/main/aust_iupc_2025

Problem A. Max-Min Madness

We begin by noting that the only thing that matters in the array is the collection of its values; that is, the positions are not important, only the multiset of numbers. In particular, if we sort the array in increasing order to get

$$a = [a_1, a_2, \dots, a_n],$$

then every operation only decreases some of these values. Moreover, if we consider the reverse-sorted array

$$\text{rev}(a) = [a_n, a_{n-1}, \dots, a_1],$$

one can prove that each valid operation strictly decreases $\text{rev}(a)$ in the lexicographic order. This fact guarantees that the process terminates after a finite number of steps.

To maximize the total number of operations, we need to “slow down” the decrease. The following greedy strategy is optimal:

- If $a_1 > 0$ (i.e. the smallest element is positive), simply perform Operation 1 on a_1 by decrementing it.

- Otherwise, when $a_1 = 0$, let i be the smallest index such that $a_i > 0$. Then perform Operation 2 on the indices $1, 2, \dots, i$. This operation sets every a_j for $1 \leq j \leq i$ to $a_i - 1$, which is exactly the lexicographic predecessor of the current a .

This strategy is optimal because every operation moves the state to its immediate predecessor in the lexicographic order (when considering the reversed array), so no extra operations can be squeezed in between.

To count the maximum number of operations, let's analyze the scenario when all elements of the array are equal.

Let $f(i, j)$ denote the maximum number of operations that can be performed on an array of length i where every element is equal to j . Basically the number of operations to convert all elements to 0.

To construct the transition, we can consider the following:

- First we need to convert all elements of the first $i - 1$ positions to 0. So this will take $f(i - 1, j)$ operations.
- Then we perform Operation 2 on the indices $1, 2, \dots, i$. This will convert all elements to $j - 1$. So this will add 1 operation.
- Now we need to convert all elements to 0. So this will take $f(i, j - 1)$ operations.

So we have the recurrence:

$$f(i, j) = f(i - 1, j) + f(i, j - 1) + 1.$$

It is convenient to define $g(i, j) = f(i, j) + 1$.

Then, substituting $f(i, j) = g(i, j) - 1$ into the recurrence gives:

$$\begin{aligned} g(i, j) - 1 &= [g(i - 1, j) - 1] + [g(i, j - 1) - 1] + 1 \\ &= g(i - 1, j) + g(i, j - 1) - 2 + 1 \\ &= g(i - 1, j) + g(i, j - 1) - 1. \end{aligned}$$

Now, add 1 to both sides:

$$g(i, j) = g(i - 1, j) + g(i, j - 1).$$

The base case is $g(0, 0) = f(0, 0) + 1 = 0 + 1 = 1$. Similarly, $g(i, 0) = g(0, i) = 1$ for all $i \geq 0$.

But this is exactly the Pascal's Triangle (rotated by 45 degrees).

Hence, we have

$$g(i, j) = \binom{i+j}{i}.$$

Or you can think of it like this: the above recurrence is the exact same recurrence of the number of ways to go from point $(0, 0)$ to point (i, j) if we can only go right or up. And that problem can be solved using stars and bars by imagining i stars and j bars and the number of such sequences is $\binom{i+j}{i}$.

So, the maximum number of operations on an array of length i with all entries equal to j is given by

$$f(i, j) = \binom{i+j}{i} - 1.$$

Using this combinatorial formula, we can compute the answer for a given array. The steps are the following:

- Sort the array in increasing order.

- First, we need to convert a_1 to 0. This will take $f(1, a_1)$ operations.
- Then, use one Operation 2 on indices 1, 2 to set both elements to $a_2 - 1$.
- Then, perform $f(2, a_2 - 1)$ operations to convert the prefix of size 2 to 0.
- Then, use one Operation 2 on indices 1, 2, ..., 3 to set all elements to $a_3 - 1$.
- Then, perform $f(3, a_3 - 1)$ operations to convert the prefix of size 3 to 0.
- Continue this process until the whole array is converted to 0.

The total number of operations is

$$f(1, a_1) + [1 + f(2, a_2 - 1)] + [1 + f(3, a_3 - 1)] + \cdots + [1 + f(n, a_n - 1)].$$

which is equivalent to

$$a_1 + \sum_{i=2}^n (f(i, a_i - 1) + 1) = a_1 + \sum_{i=2}^n \binom{i + a_i - 1}{i} - 1 + 1 = a_1 + \sum_{i=2}^n \binom{i + a_i - 1}{i}.$$

By precomputing factorials and inverse factorials, you can compute the answer easily in $O(n + M)$ time where M is the maximum element in the array.

Problem B. The Absolute MEX Challenge

We want to construct a permutation p of length n such that the set $\{|p_1 - 1|, |p_2 - 2|, \dots, |p_n - n|\}$ has a MEX (minimum excluded value) of at least $n - 1$. This means that the differences must cover all integers from 0 to $n - 2$.

First, we force a difference of 0 by setting $p_n = n$, since then $|p_n - n| = |n - n| = 0$.

Now, we can obtain the differences from $n - 2$ down to 1. One way to achieve this is as follows:

- For a difference of $n - 2$, set $p_{n-1} = 1$. Then $|p_{n-1} - (n - 1)| = |1 - (n - 1)| = n - 2$.
- For a difference of $n - 3$, set $p_2 = n - 1$. Then $|p_2 - 2| = |(n - 1) - 2| = n - 3$.
- For a difference of $n - 4$, set $p_{n-2} = 2$. Then $|p_{n-2} - (n - 2)| = |2 - (n - 2)| = n - 4$.
- For a difference of $n - 5$, set $p_3 = n - 2$. Then $|p_3 - 3| = |(n - 2) - 3| = n - 5$.

Continuing in this alternating fashion, we effectively append to the right in increasing order (assigning 1, 2, 3, ...) and to the left in decreasing order (assigning $n - 1, n - 2, \dots$) until they meet in the middle.

So basically in the following format:

$$?, n - 1, n - 2, n - 3, \dots, \dots, 3, 2, 1, n$$

Here by ? mark, it means we can set p_1 to the number that is left after filling the remaining indices as this index doesn't play a role in this construction.

This arrangement guarantees that the differences obtained contains all of $\{0, 1, 2, \dots, n - 2\}$, so the MEX is at least $n - 1$.

Thus, this construction yields a valid permutation meeting the required condition.

Problem C. Palindromic Palindrome Partition

We start by considering a simpler counting problem: How many ways can we partition a string into palindromic substrings such that not only is each substring a palindrome, but the sequence of their lengths is also a palindrome?

The key observation is that such a partition must be symmetric. In a valid partition into substrings $t_1, t_2, \dots, t_k$, the condition on the lengths forces $|t_1| = |t_k|$, $|t_2| = |t_{k-1}|$, and so on. This symmetry suggests a natural recursive DP formulation where we peel off matching palindromic pieces from both ends.

The approach is as follows:

- Fix a palindromic substring at the beginning of the string. Because the sequence of lengths must be a palindrome, the substring taken at the beginning forces a palindromic substring at the end of the string with the same length.
- Remove these two substrings and then recursively partition the remaining middle part.

In other words, if we fix the left part (say, $s[l \dots i]$) as a palindrome, then the corresponding right part ($s[j \dots r]$) is also determined (with j chosen so that the length equals that of $s[l \dots i]$). This fixed jump from both ends ensures that the lengths of the outer segments match.

So for $s[l \dots r]$, the transition is to loop over all possible left palindromes $s[l \dots i]$ and then check if $s[j \dots r]$ is a palindrome with the same length.

It looks like the complexity is $O(n^3)$, but notice that if you fix l , then the r is also fixed because you are peeling off the same length from both ends. So the complexity is actually $O(n^2)$.

Finally, to check if a substring $s[l \dots r]$ is a palindrome efficiently, we can use hashing or a simple DP approach as follows:

- Define a DP array $dp_{l,r}$ where $dp_{l,r} = 1$ if $s[l \dots r]$ is a palindrome, and 0 otherwise.
- The base case is $dp_{l,l} = 1$ for all l .
- The DP transition is $dp_{l,r} = (s[l] == s[r]) \wedge dp_{l+1,r-1}$.

Now, let's move to the scoring aspect. Suppose that the inner partition (i.e., the partition of the substring after removing the matching outer palindromes) has a score corresponding to x^2 where x is the number of segments. When we add a matching pair of palindromic substrings to the ends, the number of segments increases by 2. Notice that:

$$(x+2)^2 = x^2 + 4x + 4.$$

Thus, if the inner partition contributes a term x^2 (from the square of its segment count), adding two segments contributes three parts:

- The original x^2 ,
- A linear term $4x$, and
- A constant 4.

In our DP, we therefore need to track three components: the sum of squares (the actual scores), the linear contribution (which arises when the number of segments increases), and the count of partitions.

The overall time complexity is $O(n^2)$.

Problem D. Strong Tree

Disjoint Set Union Solution:

We will maintain disjoint sets of connected vertices in the tree. Since it is a rooted tree, each of these components will have a single vertex as its own root. We need to store the maximum possible *strength* of a path from this root to any of the vertices in the component. Let's call this value the "best-path" of the component.

Initially, each of the vertices of the tree are in separate sets. We will iterate over the vertices in ascending order of their ranks. Now for the current vertex, check all of its adjacent children. If a child has a lower rank, we can consider the best-path from its component as a candidate. We can then choose up to two candidates which have the maximum values to get the answer for our current vertex.

Afterwards, we need to merge the current vertex with the components of its adjacent vertices (including the parent) which have lower ranks. The merging of components can be done in $O(n \log n)$ using DSU. We will pre-calculate the *strength* of the paths from vertex-1 to every vertex, let's call this the "prefix-sum". You can notice that the best-path of any component is to the vertex with the maximum prefix-sum. The value of the best-path can thus be computed by subtracting the prefix-sum of the parent of the component-root from the maximum prefix-sum of any vertex in the component.

Segment Tree Solution:

Let's maintain an array of length n with initial values of 0. Consider a very large value Inf and for each vertex from 1 to n , add $-Inf$ to the array for each vertex in its corresponding sub-tree. Then we will iterate over the vertices in ascending order of their ranks. For the current vertex u , add $Inf + a_u$ to each vertex in its corresponding sub-tree. What this essentially does is, the array keeps the current *strength* of the paths from vertex-1 to every vertex, where the value of the vertices with higher ranks are replaced with Inf .

Now by checking all of its adjacent children, we can pick the highest value of a path from their subtrees. We need to subtract the value up to the current vertex to get the accurate *strength* of the path and consider it a candidate if it has a positive value. Since the vertices with higher ranks still contain $-Inf$ as their value, all of the paths that go through them will produce a negative value. We can then choose up to two candidates which have the maximum values to get the answer for our current vertex.

To efficiently perform the sub-tree operations, we can use the DFS order of a rooted tree to convert the tree into an array and get a sub-array range for each vertex and its sub-tree. Then we can use a segment-tree on this array in $O(n \log n)$ for the addition/subtraction updates and the range-maximum queries.

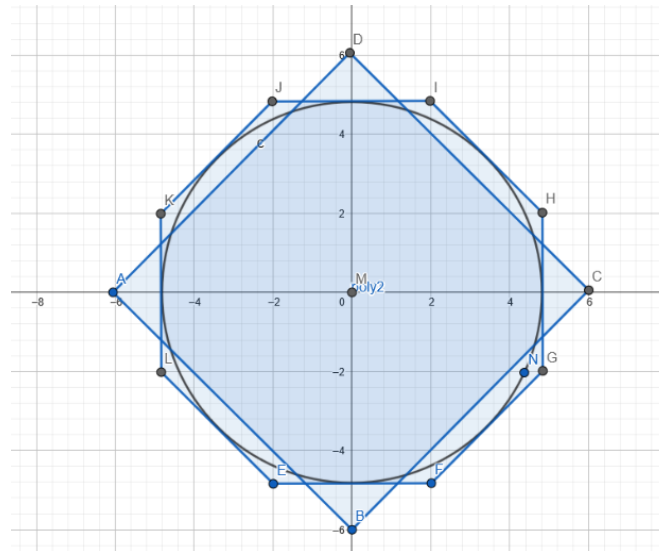
Problem E. Aloy and the Forbidden Code

The answer is always 3. Why?

WLOG, consider a string like `ababab...`. Since all three characters must be present, the third character `c` will appear at some point. When this happens, it will form a segment where the three consecutive characters are distinct.

Thus, the answer is always 3.

Problem F. Rotating Painter



We notice that some parts of the lower polygon are covered by the top polygon near its vertices. However, if we rotate the top polygon, some of these obstructed areas become visible. We can continue this process, but no matter how we rotate the top polygon, we cannot reveal any area covered by its incircle. Effectively, we replace the upper polygon with its incircle.

To determine the uncovered area, we must compute the area of the sections of the lower polygon not covered by the incircle. We do this as follows:

- The polygon is divided into the $2n$ triangle.
- We find the area of one of these triangles.
- By symmetry, we calculate the total uncovered area.
- We subtract this from the total area of the lower polygon.

So, the problem reduces to a triangle circle intersection problem. We can find the area using trigonometry and some geometry knowledge.

Let:

- a = apothem of the lower polygon,
- r = radius of the incircle = apothem of the upper polygon,
- d = circumradius of the lower polygon,
- s = side length of the lower polygon.

Here, An apothem of a polygon is a line segment drawn from the center of the polygon perpendicular to the midpoint of one of its sides; essentially, it is the shortest distance from the center of the polygon to any of its side.

The circumradius of a polygon is the radius of its circumcircle. The circumcircle of a regular polygon is the circle that passes through every vertex of the polygon.

The uncovered area depends on the relationship between r and other parameters:

- If $r \leq a$, then:

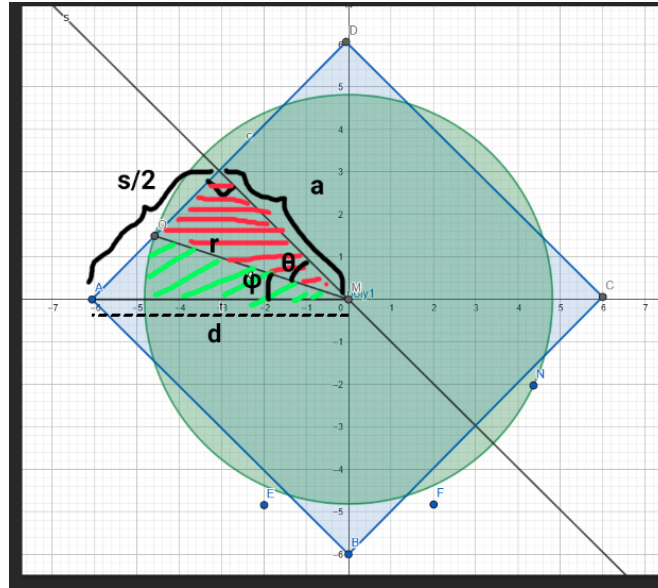
$$\text{Uncovered Area} = \frac{1}{2} \times s \times a \times n - \pi r^2. \quad (1)$$

- If $r \geq d$, then:

$$\text{Uncovered Area} = 0.$$

(2)

- Otherwise, a more detailed computation is required.



$$\text{Area of the red triangle} = \frac{1}{2} \times \sqrt{r^2 - a^2} \times a$$

$$\text{Area of the green arc} = \varphi \times \frac{r^2}{2}$$

And,

$$\varphi = \frac{2\pi}{n} - \theta$$

$$\Rightarrow \varphi = \frac{\pi}{n} - \theta$$

Since,

$$\cos \theta = \frac{a}{r}$$

$$\Rightarrow \theta = \cos^{-1} \frac{a}{r}$$

$$\therefore \varphi = \frac{\pi}{n} - \cos^{-1} \frac{a}{r}$$

Therefore, area of the arc

$$\text{Area} = \left(\frac{\pi}{n} - \cos^{-1} \frac{a}{r} \right) \times \frac{r^2}{2}$$

And total covered area is

$$= 2 \times n \times \left(\frac{1}{2} \times \sqrt{r^2 - a^2} \times a + \left(\frac{\pi}{n} - \cos^{-1} \frac{a}{r} \right) \times \frac{r^2}{2} \right)$$

So, the final answer is

$$= \frac{1}{2} \times s \times a \times n - 2 \times n \times \left(\frac{1}{2} \times \sqrt{r^2 - a^2} \times a + \left(\frac{\pi}{n} - \cos^{-1} \frac{a}{r} \right) \times \frac{r^2}{2} \right)$$

You can find the solution in multiple ways. However, the author thinks this approach is easier to understand.

Problem G. GCD and LCM in Perfect Sync

We start by observing that the condition

$$\frac{\text{lcm}(a_1, a_2, \dots, a_n)}{\text{gcd}(a_1, a_2, \dots, a_n)} = a_1$$

is multiplicative. Because lcm, gcd and division are all multiplicative.

So this can be handled one prime at a time. Write

$$a_1 = \prod_p p^{e_p},$$

so that for each prime p dividing a_1 the exponent in a_1 is e_p . For any sequence $\{a_1, a_2, \dots, a_n\}$, let f_i denote the exponent of p in a_i . Notice that $f_1 = e_p$.

The gcd and lcm for the prime p are given by

$$\text{gcd}(a_1, \dots, a_n) = p^{\min\{f_1, \dots, f_n\}}, \quad \text{lcm}(a_1, \dots, a_n) = p^{\max\{f_1, \dots, f_n\}},$$

so the contribution of p to

$$\frac{\text{lcm}(a_1, \dots, a_n)}{\text{gcd}(a_1, \dots, a_n)}$$

is

$$p^{\max\{f_i\} - \min\{f_i\}}.$$

Since the overall condition requires that

$$\frac{\text{lcm}(a_1, \dots, a_n)}{\text{gcd}(a_1, \dots, a_n)} = a_1 = p^{e_p} \times (\text{other primes}),$$

for each prime p we must have

$$\max_{1 \leq i \leq n} f_i - \min_{1 \leq i \leq n} f_i = e_p.$$

For each prime p with exponent e_p in a_1 , let's analyze how we can choose the other exponents $f_2, f_3, \dots, f_n$ (for $n - 1$ positions).

The key observation is that for any valid sequence with $f_1 = e_p$, if we have $\max\{f_i\} - \min\{f_i\} = e_p$, then the possible ranges for $[\min\{f_i\}, \max\{f_i\}]$ (or $[m, M]$) are: $[0, e_p], [1, e_p + 1], \dots, [e_p, 2e_p]$

This gives us $e_p + 1$ possible ranges.

For each range $[m, M]$, let's try to count the number of valid sequences.

In general, we can choose each f_i (for $n - 1$ positions $f_2, f_3, \dots, f_n$) to be any value within $[m, M]$, giving $(M - m + 1)^{n-1} = (e_p + 1)^{n-1}$ possibilities.

However, this overcounts sequences that don't actually achieve both the minimum and maximum values in their range. We need to apply inclusion-exclusion.

Case 1: $m = 0$

In this case, $[m, M] = [0, e_p]$

- All $f_2, f_3, \dots, f_n$ can be chosen from $[0, e_p]$, so there are $(e_p + 1)^{n-1}$ valid sequences.
- Subtract sequences where no f_i equals the minimum:
 - The number of choices for f_i is e_p (any of $[1, e_p]$)
 - There are $n - 1$ positions to choose, so the total number of sequences is e_p^{n-1}
- Subtract sequences where no f_i equals the maximum:
 - But as $f_1 = e_p$, this is impossible as we already have f_1 as the maximum.
 - So we don't need to subtract anything here.

So the number of valid sequences in this case is:

$$(e_p + 1)^{n-1} - e_p^{n-1}$$

Case 2: $m = e_p$

In this case, $[m, M] = [e_p, 2e_p]$

- All $f_2, f_3, \dots, f_n$ can be chosen from $[e_p, 2e_p]$, so there are $(2e_p - e_p + 1)^{n-1} = (e_p + 1)^{n-1}$ valid sequences.
- Subtract sequences where no f_i equals the minimum:
 - As $f_1 = e_p$, this is impossible as we already have f_1 as the minimum.
 - So we don't need to subtract anything here.
- Subtract sequences where no f_i equals the maximum:
 - The number of choices for f_i is e_p (any of $[e_p, 2e_p - 1]$)
 - There are $n - 1$ positions to choose, so the total number of sequences is $(e_p - 1)^{n-1}$

So the number of valid sequences in this case is:

$$(e_p + 1)^{n-1} - e_p^{n-1}$$

Case 3: $0 < m < e_p$

In this case, $M = m + e_p$

- All $f_2, f_3, \dots, f_n$ can be chosen from $[m, M]$, so there are $(M - m + 1)^{n-1} = (e_p + 1)^{n-1}$ valid sequences.
- Subtract sequences where no f_i equals the minimum:

- The number of choices for f_i is $M - m$ (any of $[m + 1, M]$)
- There are $n - 1$ positions to choose, so the total number of sequences is $(M - m)^{n-1} = e_p^{n-1}$
- Subtract sequences where no f_i equals the maximum:
 - The number of choices for f_i is $M - m$ (any of $[m, M - 1]$)
 - There are $n - 1$ positions to choose, so the total number of sequences is $(M - m)^{n-1} = e_p^{n-1}$
- Add back sequences where neither minimum nor maximum appears (as these were subtracted twice):
 - The number of choices for f_i is $M - m - 1$ (any of $[m + 1, M - 1]$)
 - There are $n - 1$ positions to choose, so the total number of sequences is $(M - m - 1)^{n-1} = (e_p - 1)^{n-1}$

So the number of valid sequences for a fixed range $[m, M]$ is:

$$(e_p + 1)^{n-1} - e_p^{n-1} - e_p^{n-1} + (e_p - 1)^{n-1} = (e_p + 1)^{n-1} - 2e_p^{n-1} + (e_p - 1)^{n-1}$$

So for each $0 < m < e_p$, ($e_p - 1$ such ranges), the number of valid sequences is:

$$(e_p - 1) [(e_p + 1)^{n-1} - 2e_p^{n-1} + (e_p - 1)^{n-1}]$$

So over all three cases, the total number of valid sequences is:

$$[(e_p + 1)^{n-1} - e_p^{n-1}] + [(e_p + 1)^{n-1} - e_p^{n-1}] + (e_p - 1) [(e_p + 1)^{n-1} - 2e_p^{n-1} + (e_p - 1)^{n-1}]$$

Which is the same as (after some simplification):

$$[(e_p + 1)^n - 2e_p^n + (e_p - 1)^n]$$

The final answer is the product over all primes p dividing a_1 :

$$\text{ans} = \prod_{p|a_1} [(e_p + 1)^n - 2e_p^n + (e_p - 1)^n] \mod 998\,244\,353$$

So basically the solution is: factorize a_1 , compute the contribution for each prime factor as shown using fast exponentiation, and multiply all contributions modulo 998 244 353.

Problem H. Flip to Zero

A simple observation is that the actual arrangement of bits is irrelevant – only the total number of ones matters. In one operation, you choose exactly k indices to flip. If there are currently x ones (and $n - x$ zeros), then by flipping r ones (and therefore $k - r$ zeros) you change the count of ones to

$$x - r + (k - r) = x + k - 2r.$$

Since r can vary subject to the constraints imposed by the number of ones and zeros available, the number of ones changes in steps of 2.

We can model the problem as a graph problem where each node is a state which is an integer between 0 and n representing the number of ones. Our goal is to find the minimum number of operations to reach each state starting from the all-zero state (state 0). Since operations are reversible, finding the distance from 0 to any state m gives the minimum number of operations needed to turn a string with m ones into all 0s.

A high-level outline of the solution is as follows:

- Treat each state as the current number of ones in the string.
- Start from state 0 (all bits are 0) with distance 0.
- From any state x , an operation allows a transition to states of the form

$$x + k - 2r,$$

where r (the number of ones flipped) is chosen such that the move is valid. This gives a range of reachable states with differences changing by 2.

- Use a breadth-first search (BFS) to compute the minimum number of operations needed to reach each state.
- There might be multiple direct edges (transitions) from a node, but as all of them are in a range (partitioned by odd or even numbers), we can maintain two sets of unvisited even and odd nodes and do the transitions efficiently.
- The answer for each m (for $1 \leq m \leq n$) is the computed distance from state 0 to state m , or -1 if state m is unreachable.

This approach efficiently computes $f(m)$ for all m from 1 to n in $O(n \log n)$ time.

Problem I. The Art of Rearrangement

One possible way to ensure all values of $a_i - i$ are distinct is to sort the array in decreasing order. This works because if a is sorted in descending order, then for every i :

$$a_i - i > a_{i+1} - (i + 1)$$

This holds because $a_i \geq a_{i+1}$, and since $i < i + 1$, subtracting a larger index from a smaller value ensures that the differences strictly decrease. As a result, all values remain distinct.

Thus, sorting the array in decreasing order guarantees a valid arrangement.

Problem J. No Duplicates

Let's solve this problem step by step.

For each testcase, we need to process q restrictions one by one, and after each restriction, we need to calculate the number of valid arrays.

One key observation is that if an interval is fully covered by another interval, we can just ignore the smaller interval.

So we can think of the following approach:

- Sort all intervals by left endpoint and process them in that order.
- For each interval $[l, r]$, we only need to consider the portion that isn't already covered by previous intervals.

- Let's say for an interval of length s , a portion of length c is already covered by previous intervals.
 - The already covered portion is guaranteed to have distinct elements
 - For the remaining $(s - c)$ positions, we need to choose distinct elements different from the c elements (so there are $m - c$ choices)
 - This contributes $P(m - c, s - c)$ to our answer, where $P(n, r)$ represents permutation i.e. $P(n, r) = \frac{n!}{(n-r)!}$
- For positions not covered by any interval, each position contributes a factor of m to the answer.

You can precompute factorials and inverse factorials to speed up the permutation calculation. It can be done in $O(m)$.

Now to solve the problem with each new restriction:

- Maintain a set of active intervals (those that contribute to the answer)
- For each interval, track its contribution factor (its contribution divided by m^s)
- When adding a new interval:
 - Remove or split existing intervals that get covered by the new one
 - Update contributions accordingly
 - Add the new interval with its contribution

Implementation is a bit complex. So please be careful.

The final time complexity is $O(q(\log q + \log n) + m)$.

Problem K. Primal Brackets

We can view the input balanced sequence as a regular balanced bracket sequence where each positive number represents an opening bracket $[+x]$ and each negative number represents a closing bracket $[-x]$. Although the brackets are distinct, the overall structure is identical to that of a standard balanced parentheses string.

Imagine constructing a **bracket tree** for the sequence:

- The entire sequence is the root.
- For any node corresponding to a balanced subarray of the form $[+x] \ A \ B \ C \ \dots \ [-x]$, the balanced subarrays A , B , C , $\dots$ are its children.
- The leaf nodes of this tree are exactly the minimal balanced pairs of length 2, that is, sequences of the form $[+x, -x]$.

A key observation is that any partition of the sequence into primal subsequences must have at least as many subsequences as there are leaves in the bracket tree. This is because you cannot merge two different leaves, say $[+x, -x]$ and $[+y, -y]$, into a single primal subsequence. In a primal sequence all the positive elements must come before all the negative ones, and merging two leaves would force the closing bracket of the first pair to come before the opening bracket of the second, which is not allowed.

Thus, if we denote the number of leaves by $\#leaf$, we must have

$$f(b) \geq \#leaf.$$

It turns out that this bound is tight. One way to see this is to use a dynamic programming (DP) approach on the bracket tree:

- For a subtree corresponding to $[+x] \ A \ B \ C \ \dots \ [-x]$, assume that you have already partitioned each of its child subtrees into the minimum number of primal subsequences.
- You can then append the outer brackets $[+x]$ and $[-x]$ to one of the primal subsequences from its children, so no additional subsequence is needed for this subtree.
- The only case where you must form a new primal subsequence is when the subtree is a leaf itself (i.e. it is of the form $[+x, -x]$).

Therefore, the minimum number of primal subsequences needed is exactly equal to the number of leaves in the bracket tree.

A simpler implementation leverages the fact that a leaf appears as a consecutive pair $[+x, -x]$ in the sequence. Define an indicator variable $\text{is_leaf}_i = 1$ if the pair (a_{i-1}, a_i) forms a leaf (i.e. if $a_{i-1} = -a_i$), and 0 otherwise. Then, by building a prefix sum array p where $p_i = p_{i-1} + \text{is_leaf}_i$, the answer for any balanced subarray $a[l \dots r]$ is given by

$$f(a[l \dots r]) = p_r - p_{l-1},$$

So the final implementation is as follows:

```
for i in 1 to 2n:
    p[i] = p[i-1] + (a[i-1] == -a[i])
for i in 1 to q:
    l, r = read()
    print(p[r] - p[l-1])
```