Important Links

- **Main Contest:** https://toph.co/c/cuet-iupc-2025
- **Practice Contest:** https://codeforces.com/gym/106259
- **Team Ratings:** https://therealbcs.com/team-ratings?contest_id=cuet_iupc_2025
- **Problem Setters:** https://therealbcs.com

Problem Contributors

Problem	Author	Developer	Tester
A. Kill Two Birds...	MD Irfanur Rahman Rafio	MD Irfanur Rahman Rafio	Kawsar Hossain
B. K Floors Down	Shahjalal Shohag	Shahjalal Shohag	Al-Shahriar Tonmoy
C. Pattern Purifier	Shahjalal Shohag	Kawsar Hossain	Rudro Debnath
D. The AND, The OR...	Reyad Hossain	Shahjalal Shohag	MD Irfanur Rahman Rafio
E. A Slice of Pi	Shahjalal Shohag	Mainul Hasan	Kawsar Hossain
F. Survival of the Fated	Al-Shahriar Tonmoy	Al-Shahriar Tonmoy	Mainul Hasan
G. Pascal's Tree	Shahjalal Shohag	Al-Shahriar Tonmoy	Tasmeem Reza
H. Prime Triangles	Kawsar Hossain	Kawsar Hossain	MD Irfanur Rahman Rafio
I. Peak Reduction	Shahjalal Shohag	Rudro Debnath	Mainul Hasan
J. The Power of the Sun	Shahjalal Shohag	Mainul Hasan	Kawsar Hossain
K. The Great Withering	Shahjalal Shohag	Shahjalal Shohag, Sayeef Mahmud	Tasmeem Reza

Problem Difficulty & Statistics

Problem	Difficulty	Expected #AC	Actual #AC	Tags
I. Peak Reduction	**Div2A**	130	130	Adhoc
C. Pattern Purifier	**Div2B**	103	126 ↑	Adhoc, Strings, DS
A. Kill Two Birds...	**Div2B**	89	69 ↓	Adhoc, Graphs
D. The AND, The OR...	**Div2C**	64	119 ↑↑	Bits, Greedy
F. Survival of the Fated	**Div2C**	38	43 ↑	Expected Value
J. The Power of the Sun	**Div2D**	14	22 ↑	Data Structures
H. Prime Triangles	**Div2D**	11	0 ↓↓	Constructive, NT, Geo
E. A Slice of Pi	**Div2D**	7	2 ↓	Geometry, Two Pointers
B. K Floors Down	**Div2E**	7	1 ↓	Math, DP, Combinatorics
G. Pascal's Tree	**Div2E**	2	0 ↓	Trees, DS
K. The Great Withering	**Div2E**	2	2	DP, Graphs, Combinatorics

514	11	132	12
Total ACs	Problems	Teams	Judges

Judge Panel

Role	Name
Coordinator	Shahjalal Shohag
Contest Reviewer	Jubayer Nirjhor
Onsite Technical Director	Fahim Tajwar Saikat
VIP Tester	Alfeh Sani
Judges	Check the problem contributors table

The problemset was managed by the **BCS team**. Problems were reviewed and selected by the Coordinator and Contest Reviewer. Thanks to all **onsite judges** and **university volunteers** for their help in running the contest!

We hope that you liked the problems! Let us know if you have any feedback.

Solutions to All Problems

https://github.com/ShahjalalShohag/bcs-contest-resources/tree/main/cuet_iupc_2025

Tutorials

Problem A. Kill Two Birds with One Stone

Hint 1: Start with two simplified variations of the problem. In the first variation, assume that all the initial cells are 1. In the second one, assume that in one operation, you can decrease any two entries regardless of whether they are adjacent or not.

Hint 2: Start with small matrices.

Hint 3: Look for invariants.

Solution

1. Basic Observations

Each operation decreases two adjacent entries by 1. Since the target is to transform the given matrix into a null matrix, you can only decrease an entry if it is currently 1; if any entry becomes negative, you cannot increase it back to 0.

Denote the operation of decreasing the value of two adjacent cells from 1 to 0 as **throwing a stone** at those cells. Also, let's call cells that are initially 0 **dead cells** for brevity.

Initially, the matrix contains $(nm - 2)$ ones and 2 zeroes. Each stone changes exactly two entries from 1 to 0. So, if $(nm - 2)$ is odd (i.e., nm is odd), the task is impossible. **Answer:** `NO`.

2. One-Dimensional Case ($n = 1$ or $m = 1$)

In this case, the matrix is a single row or column. The two dead cells divide this line into three disconnected segments of ones. A stone cannot cross a dead cell.

Answer: `YES` if and only if all three segments have even size.

3. Base Two-Dimensional Cases ($\min(n, m) > 1$)

2×2 **Matrices**:

Analyzing all binary matrices yields the following results:

- `YES` if the dead cells are adjacent.
- `NO` if they lie on opposite diagonals.

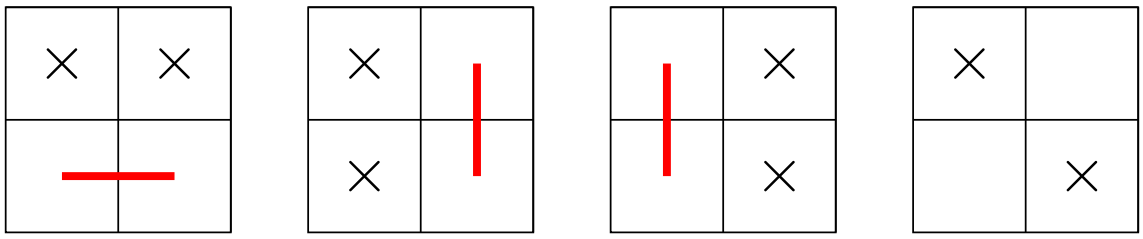

Figure: All Possible 2×2 Matrices

2×3 **Matrices**:

Checking all possibilities shows the following results:

- `NO` if the Manhattan distance of the dead cells is 2.
- `YES` otherwise.

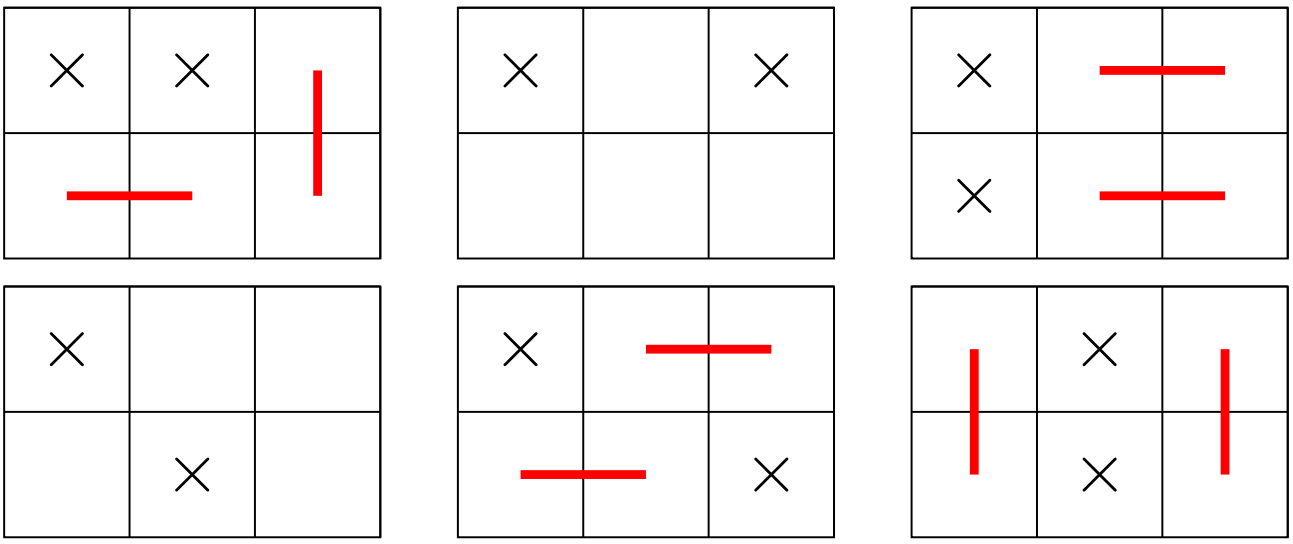

Figure: All Possible Distinct 2×3 Matrices up to Reflection

This rule is a generalized version of the result from 2×2 matrices, and it also applies to 3×2 matrices.

4. Extending to Larger Matrices

It can be proven that inserting two rows or two columns of ones anywhere in the matrix does not change the answer.

- If added at the boundary, they form a clean rectangle of ones that can be removed independently.

- If inserted inside the matrix, any stone crossing the insertion in the original solution can be split into two stones.

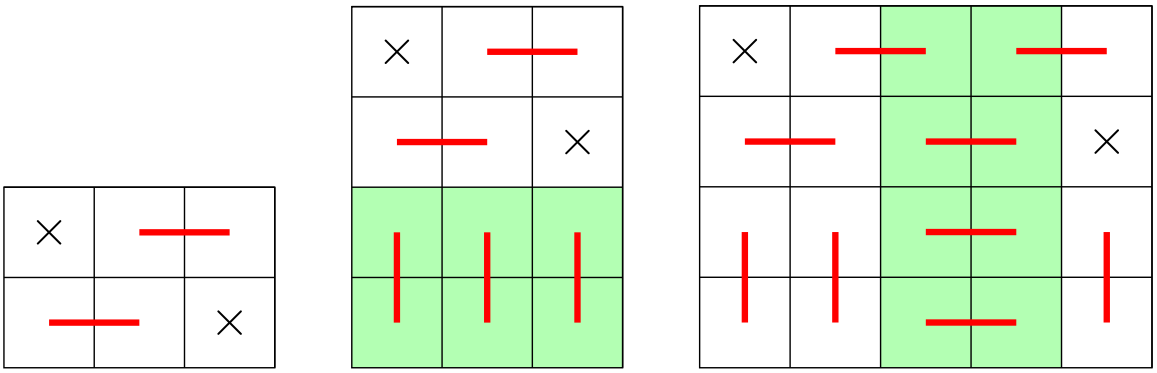

Figure: Inserting Two New Rows or Columns – Either at the Boundary or Internally (Shown in Green) – Maintains the Nullification Strategy and Parity of Manhattan Distances

Starting from small base cases and repeatedly adding two rows or columns allows building any matrix with $\min(n, m) > 1$. Depending on whether the new rows/columns are inserted between the dead cells or not, the Manhattan distance either increases by 2 or stays the same.

Thus, you can generalize for all $2D$ matrices (matrices with at least two rows and at least two columns):

A matrix can be nullified *iff* the Manhattan distance between the two dead cells is odd.

Alternate Solution

Handle trivial cases (nm is odd, $n = 1$, $m = 1$) separately.

1. Discovering Invariant

Color the grid like a chessboard:

- Even distance from top-left → white
- Odd distance → black

A stone always hits one white and one black cell. The matrix has an equal number of white and black cells.

- If both dead cells have the same color: imbalance arises → **NO**.
- If dead cells have opposite colors: equivalent to Manhattan distance being odd → **YES**.

2. Constructive Proof via Hamiltonian Circuit

Without loss of generality, you can assume that n is even (otherwise transpose the matrix). Transposition preserves adjacency rules and the Manhattan distance between any two cells.

When n is even, build a Hamiltonian circuit:

- Start at bottom-left, traverse bottom row left to right.
- From bottom-right, move up the rightmost column to the top row.

- Traverse top row right to left, completing the boundary.
- For each interior row i $(2 \leq i \leq n-1)$:
  - If i is even: traverse left to right up to column $m-1$.
  - If i is odd: traverse right to left starting at column $m-1$.

This forms a zigzag interior path. Since n is even, row $n-1$ is traversed right to left. Connect all traversed segments to form a single Hamiltonian circuit visiting every cell exactly once.

Because the dead cells have opposite colors, the number of cells between them along the circuit is always even, regardless of direction. To place stones:

- Choose a dead cell and choose its 'next' cell as the starting cell.
- Walk along the circuit, pairing cells two at a time.
- Skip a dead cell when encountered.
- Each valid pair contains only usable cells.

This yields a full set of valid stone placements.

An example for the fifth sample:

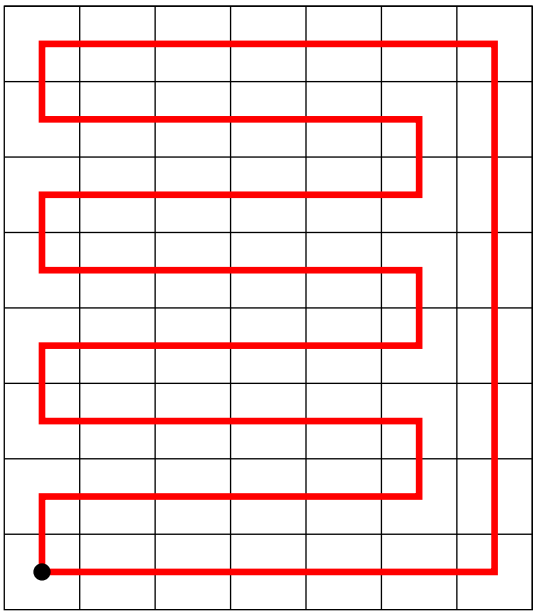

Figure: Hamiltonian Circuit for Test Case 5

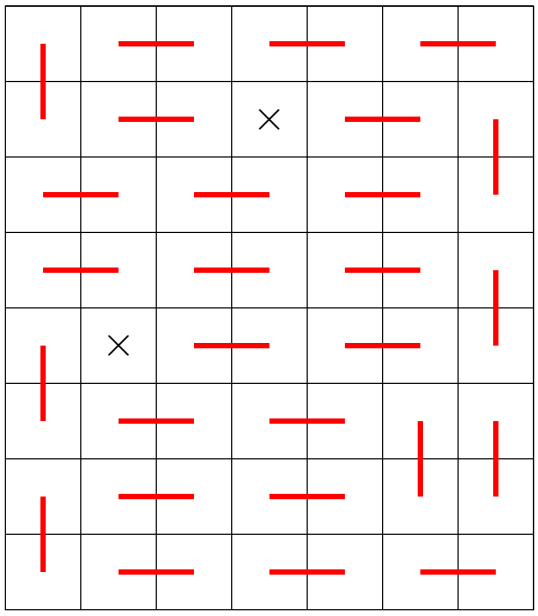

Figure: Solution Construction for Test Case 5

Note: When $n = 1$ or $m = 1$, the underlying graph is a line, so a Hamiltonian circuit cannot exist.

Problem B. K Floors Down

1. The Floor Identity

Before diving into the solution, we must prove a key property of the floor function:

$$\left\lfloor \frac{\lfloor x/y \rfloor}{z} \right\rfloor = \left\lfloor \frac{x}{y \cdot z} \right\rfloor$$

Proof: Let $x = q(yz) + r$, where $0 \le r < yz$. By definition, $\lfloor x/(yz) \rfloor = q$. We can rewrite r using division by y: $r = a \cdot y + b$, where $0 \le b < y$. Since $r < yz$, it follows that $a < z$. Substituting back:

$$x = q(yz) + ay + b = (qz + a)y + b$$

First, calculate the inner floor $\lfloor x/y \rfloor$:

$$\lfloor x/y \rfloor = \lfloor (qz + a) + b/y \rfloor = qz + a \quad (\text{since } 0 \le b/y < 1)$$

Next, divide this result by z and take the floor:

$$\left\lfloor \frac{qz + a}{z} \right\rfloor = \left\lfloor q + \frac{a}{z} \right\rfloor = q \quad (\text{since } 0 \le a < z)$$

Both sides equal q. Thus, the operations are equivalent.

2. Simplifying the Problem

Using the identity above, we can unroll the recursive definition of the sequence contribution:

$$t_1 = x_1$$
$$t_2 = \lfloor x_1/x_2 \rfloor$$
$$t_3 = \lfloor \lfloor x_1/x_2 \rfloor / x_3 \rfloor = \lfloor x_1/(x_2 \cdot x_3) \rfloor$$

By induction, the final term is:

$$t_k = \left\lfloor \frac{x_1}{x_2 \cdot x_3 \cdot \ldots \cdot x_k} \right\rfloor$$

Let $P = x_2 \cdot x_3 \cdot \ldots \cdot x_k$ be the product of the last $k-1$ terms. Our goal is to sum $\lfloor x_1/P \rfloor$ over all valid sequences.

3. Key Observation: Product Constraint

We only care about cases where $P \le n$. If $P > n$, then $\lfloor x_1/P \rfloor = 0$ for all valid x_1 (since $x_1 \le n$), contributing nothing to the answer. Since the smallest integer greater than 1 is 2, the maximum number of elements greater than 1 we can multiply before exceeding n is $\log_2 n$.

- We can pick at most $m \approx \log_2 n$ elements that are > 1.
- The remaining $k - 1 - m$ elements in the denominator must be 1.

4. Dynamic Programming

We need to count how many ways we can form a product P using exactly m elements (all > 1). Let $cnt[v]$ be the frequency of number v in the input array a. Let $dp[v]$ be the number of ways to form product v using the current number of elements.

Base Case: Initially, we have chosen 0 elements. The product is 1.

$$dp[1] = 1, \quad dp[v] = 0 \text{ for } v > 1$$

Transitions: We iterate m from 1 to $\min(k-1, \log_2 n)$. In each step, we calculate a new state dp_{new} from dp_{old}.

$$dp_{new}[i \cdot j] = \sum dp_{old}[i] \times cnt[j]$$

Here, i is an existing product, and j is the next chosen number ($j \geq 2$). The constraints are $i \cdot j \leq n$.

Combinatorics: For a fixed length m and product val, the number of full sequences $(x_2, \ldots, x_k)$ is:

$$ways(val, m) = dp[val] \times \binom{k-1}{m} \times (cnt[1])^{k-1-m}$$

We sum these up into a total frequency array $tot[val]$:

$$tot[val] = \sum_{m} ways(val, m)$$

5. Final Calculation

Now we have $tot[v]$, which tells us how many sequences produce a denominator product of v. The final answer is:

$$\sum_{v=1}^{n} \left(tot[v] \times \sum_{x_1 \in a} \lfloor x_1/v \rfloor \right)$$

Calculating the inner sum naively is slow. Instead, note that $\lfloor x_1/v \rfloor$ is constant for x_1 in ranges $[c \cdot v, (c+1) \cdot v - 1]$. We iterate through multiples of v:

$$\text{contribution}(v) = tot[v] \times \sum_{j=1}^{\lfloor n/v \rfloor} j \times (\text{count of } x_1 \in [j \cdot v, (j+1) \cdot v - 1])$$

We can compute the count in $O(1)$ using a prefix sum array of frequencies.

Complexity Analysis

- **DP Step:** We iterate m up to $\log n$. In each iteration, we process products up to n. The transition behaves like a harmonic series sum. Complexity: $O(n \log n \cdot \log n)$ or simply $O(n \log^2 n)$.
- **Final Sum:** We iterate multiples for every v. This is the standard harmonic series: $n/1 + n/2 + \cdots + n/n = O(n \log n)$.
- **Total Time Complexity:** $O(n \log^2 n)$.

Problem C. Pattern Purifier

You are allowed to delete any even-length palindromic substring. Every such palindrome has two equal central characters, so it can always be removed completely by repeatedly deleting that adjacent equal pair. For example, the palindrome "`redder`" can be removed as (where the underlined substring is the one being removed and ε denotes the empty string):

$$\underline{redd}er \;\to\; r\underline{ee}r \;\to\; \underline{rr} \;\to\; \varepsilon.$$

So now we don't have to worry about any even-length palindrome; the problem has been converted to only removing adjacent equal pair elements!

For repeatedly deleting adjacent equal pairs, this process can be simulated with a stack: traverse the string from left to right; if the stack is nonempty and its top equals the current character, pop it (removing a pair), otherwise push the character. This greedy pair-removal matches any valid sequence of allowed deletions.

If the stack is empty after processing the whole string, the entire string can be deleted; otherwise it cannot. The algorithm runs in $O(|s|)$ time and $O(|s|)$ memory.

Problem D. The AND, The OR, and The XOR

The Bitwise Property

Let's analyze the formula bit-by-bit. For any specific bit position (say, the k-th bit), look at the values of that bit across all numbers in your chosen subsequence. There are only three scenarios:

- **All bits are 0:** The AND is 0, the OR is 0. Result: $0 \oplus 0 = 0$.
- **All bits are 1:** The AND is 1, the OR is 1. Result: $1 \oplus 1 = 0$.
- **Mixed (some 0, some 1):** The AND is 0 (because not **all** are 1), the OR is 1 (because **at least** one is 1). Result: $0 \oplus 1 = 1$.

Conclusion: The score of a subsequence is simply the count of bit columns that contain **both** a 0 and a 1.

Why Size 2 is Optimal

Since the score counts *mixed* columns, let's see what happens when we increase the subsequence size. If a column is already mixed (contains a 0 and a 1), adding another number cannot make it *un-mixed.* It will still contain both. However, if a column is currently *pure* (all 0s or all 1s), adding a new number might introduce a conflict, turning that column from a 0 contribution to a 1.

Therefore, adding elements can only **increase** (or maintain) the score. To minimize the score, we should pick the smallest possible subsequence size allowed, which is **2**.

Reduction to XOR

For a subsequence of exactly two numbers, a and b, a bit column is *mixed* if and only if the bits differ (0 and 1). This is exactly the definition of the bitwise XOR operation.

$$\text{score}(a, b) = a \oplus b$$

The problem is now reduced to finding the minimum XOR of any pair in the array.

The Sorting Strategy

Claim: After sorting the array ($a_1 \leq a_2 \leq \cdots \leq a_n$), the minimum XOR pair is always adjacent.

Proof: Suppose for contradiction that the minimum XOR is achieved by a_i and a_k where $i < k$ and they are not adjacent (i.e., there exists some j with $i < j < k$).

Let b be the highest bit position where a_i and a_k differ. Since $a_i < a_k$, we have: a_i has 0 at bit b, and a_k has 1 at bit b.

Since $a_i \leq a_j \leq a_k$, the value a_j has either 0 or 1 at bit b:

- If a_j has 0 at bit b (same as a_i): Then $a_i \oplus a_j$ has 0 at bit b and at all higher bits (since a_i, a_j, a_k all agree above bit b). So $a_i \oplus a_j < 2^b \leq a_i \oplus a_k$.
- If a_j has 1 at bit b (same as a_k): Then $a_j \oplus a_k$ has 0 at bit b and at all higher bits. So $a_j \oplus a_k < 2^b \leq a_i \oplus a_k$.

In either case, we found a pair with smaller XOR than $a_i \oplus a_k$, contradicting our assumption. Therefore, the minimum XOR pair must be adjacent.

Algorithm: Sort the array and find $\min(a_i \oplus a_{i+1})$ for all i.

Alternative (Trie)

You could also solve this using a **Binary Trie**. You would insert numbers one by one and, for each number, query the Trie for the value that minimizes XOR (by trying to match bits). However, a Trie is overkill here. It requires $O(n \cdot \log(\max A))$ time and significantly more lines of code. Sorting is $O(n \log n)$ and much simpler to implement.

Problem E. A Slice of Pi

Angle Conversion and Merging Collinear Points:

We begin by converting each topping (x_i, y_i) into its polar angle, and then sorting all toppings by their angles. During this process, whenever multiple toppings lie exactly on the same ray from the origin (that is, they share the same polar angle), we merge them into a single point whose tastiness value is the sum of their values. This is correct because any slice is determined entirely by its boundary angles, so toppings on the same ray are either always all included or always all excluded together; merging them therefore preserves every possible slice outcome.

After this merging, we obtain a one-dimensional list of angles with associated values. Since the pizza is circular, a slice may wrap around past angle 2π and return toward 0. To simulate this wrap-around in a linear structure, we duplicate the entire sorted list and add 2π to the angles in the second copy. Concatenating these two lists allows every circular interval of width at most one full revolution to appear as a contiguous segment in the doubled list.

Area Constraint Reformulated as an Angular Constraint:

A slice of area at most A corresponds to an angular width not exceeding $\theta_{\max} = \frac{2A}{R^2}$. Thus the problem becomes the following: in the doubled angle list, find a segment whose angular span is at most $\theta_{\max}$ and whose total tastiness value has the largest possible absolute value.

Finding the Maximum Absolute Subarray Sum Using Prefix Sums:

To handle this, we use prefix sums combined with a sliding window. Whenever the window is positioned so that the angular difference between its endpoints does not exceed $\theta_{\max}$, we want to determine which subarray inside this window gives the largest absolute sum. Any subarray sum can be expressed as $P_j - P_i$ for two prefix sums P_i and P_j with $i < j$. Inside a fixed window, consider all prefix sums whose indices lie within it. Among these numbers, let U denote the largest prefix sum and let L denote the smallest prefix sum. For any two prefix sums inside the window we always have $P_j - P_i \leq U - L$, and this upper bound is actually achieved by choosing $P_j = U$ and $P_i = L$. Similarly, the most negative subarray sum is $L - U$, whose absolute value is also $U - L$. Thus the greatest possible absolute subarray sum inside the window is simply the difference $U - L$.

Maintaining Extremal Prefix Sums Efficiently:

By sliding the window across the doubled array and maintaining the values U (largest prefix sum inside the window) and L (smallest prefix sum inside the window), we obtain the maximum possible flavor intensity under the area constraint. To maintain this range efficiently during the window expansion and contraction, we may store all valid prefix sums inside a `multiset`, allowing quick retrieval of the smallest and largest elements, giving an overall time complexity of order $n \log n$.

Alternative solution:

For every prefix sum, let j be the rightmost position inside the window where the maximum positive subarray ends; then the contribution is $P_j - P_i$ where P_i is the smallest prefix sum in that range. If we

maintain two arrays simultaneously—one with the original prefix sums and one with all values negated—we can track both positive and negative contributions at the same time, obtaining the same extremal difference.

Problem F. Survival of the Fated

Let $A = \{a_1, a_2, \ldots, a_n\}$ be the given array. Let S denote the total score after performing all $n-1$ operations. We aim to compute $E[S]$, the expected value of S.

Linearity of Expectation:

Let X_k denote the score contributed in the k-th operation. By the linearity of expectation, we have

$$E[S] = E\Big[\sum_{k=1}^{n-1} X_k\Big] = \sum_{k=1}^{n-1} E[X_k].$$

This property holds regardless of dependencies between operations, including the randomness of which element is deleted in each step.

Contribution of a Pair:

Consider an unordered pair (a_i, a_j), with $i < j$. Let I_{ij} be the indicator random variable which equals 1 if this pair is chosen in some operation and 0 otherwise. The contribution of this pair to the score when chosen is $|a_i - a_j|$. Then

$$E[S] = \sum_{i<j} E[I_{ij}] \cdot |a_i - a_j|.$$

Due to the symmetry of the process and uniform random selection and deletion, the expected contribution of each pair (i, j) is identical.

Combinatorial Weight:

There are $\binom{n}{2}$ unordered pairs in total, and exactly $n-1$ operations occur. By symmetry, the expected number of times a particular pair (a_i, a_j) contributes is

$$E[I_{ij}] = \frac{n-1}{\binom{n}{2}} = \frac{2}{n},$$

since

$$\binom{n}{2} = \frac{n(n-1)}{2} \implies \frac{n-1}{\binom{n}{2}} = \frac{2}{n}.$$

Expected Final Score:

Substituting $E[I_{ij}]$ into the sum, we obtain

$$E[S] = \sum_{i<j} |a_i - a_j| \cdot \frac{2}{n} = \frac{2}{n} \sum_{1 \le i < j \le n} |a_i - a_j|.$$

This formula gives the expected final score exactly, independent of the order of deletions or the particular values in the array. The total calculation can be done in $O(n)$ complexity using prefix sum.

Problem G. Pascal's Tree

We are given a rooted tree with n nodes and a permutation p of length n. We define S_0 as the sum of p, and generate sequences $S_1, S_2, \ldots, S_{n-1}$ by repeatedly replacing each adjacent pair with its Lowest Common Ancestor (LCA). The task is to compute the sum of each sequence S_k.

Let us define

$$f_i = \mathrm{LCA}(p_i, p_{i+1}), \quad 1 \le i \le n-1$$

and

$$d_i = \mathrm{depth}(f_i).$$

Key Observation: In the next transformation, $\text{LCA}(f_j, f_{j+1})$ is always either f_j or f_{j+1}, specifically the one with smaller depth (shallower).

Proof: Since $f_j = \text{LCA}(p_j, p_{j+1})$, the node f_j is an ancestor of both p_j and p_{j+1}. Similarly, $f_{j+1} = \text{LCA}(p_{j+1}, p_{j+2})$ is an ancestor of both p_{j+1} and p_{j+2}.

Therefore, both f_j and f_{j+1} are ancestors of p_{j+1}. Since the ancestors of any node form a chain from that node to the root, one of f_j and f_{j+1} must be an ancestor of the other. Thus $\text{LCA}(f_j, f_{j+1})$ equals the shallower one (smaller depth).

So that means, in all the following transformations, there will be no new nodes added to the sequence S_k except the nodes that are present in the sequence S_{k-1}. So each node in S_k for any $k > 1$, will be present in the sequence S_1! (Note that here we are using f as S_1 for simplicity.)

For each position k, define l_k as the number of consecutive nodes to the left of f_k with depth strictly greater than d_k, and r_k as the number of consecutive nodes to the right with depth greater than or equal to d_k. Then f_k contributes to the sums $S_1, \ldots, S_{l_k+r_k-1}$.

The contribution weight of f_k follows a symmetric (triangular) pattern. Specifically, let $m = \min(l_k, r_k)$:

- For $i = 1$ to m, the weight is i.
- For $i = l_k + r_k - m + 1$ to $l_k + r_k - 1$, the weight decreases from m down to 1.
- For the middle layers, the weight remains constant at m.

This pattern can be efficiently applied to all sequences using a difference array and iterate $2 \times m$ times for first and second point.

The algorithm is as follows:

1. Preprocess LCA and depth of every node using binary lifting in $O(n \log n)$.
2. Compute f_i and d_i for all i.
3. Compute l_i and r_i using monotonic stacks to find nearest smaller elements. This takes $O(n)$.
4. For each f_i, apply its triangular contribution to a difference array.
5. Take the prefix sum of the difference array to obtain the sums of $S_1, \ldots, S_{n-1}$. S_0 is simply the sum of p.

Correctness follows from the fact that f_k survives in the cascade exactly while its depth is not exceeded by neighbors, and the triangular contribution accounts for the increasing and decreasing multiplicities across layers.

The time complexity is $O(n \log n)$ due to LCA preprocessing, with all other steps linear in n.

Problem H. Prime Triangles

We need to construct N triangles with **distinct prime areas** using at most $\lceil \frac{N}{4} \rceil + 6$ lattice points. This means each added point should ideally produce 4 different prime areas.

Geometric Foundation:

A triangle with base 2 has area equal to its height:

$$\frac{1}{2} \times 2 \times h = h$$

Thus, to get a triangle of prime area P, we only need a point at height P from a fixed base of length 2.

Naive Approach ($\approx N$ points):

Fix a base at $(0,0)$ and $(0,2)$. For any prime P, placing the point $(P,0)$ forms a triangle of area exactly P. Each point contributes one prime area.

Twin-Primes Trick($\approx N/2$ points):

Fix points $(0,0),\ (0,2),\ (2,0)$. Now consider a prime P such that P and $P-2$ form a twin-prime pair, and add the point $(P,0)$.

- Triangle $(0,0),\ (0,2),\ (P,0)$ has area P.
- Triangle $(0,2),\ (2,0),\ (P,0)$ has area $P-2$.

Dual-Axis Strategy ($\approx N/4$ points):

Fix four base points: $A\ (0,0),\ B\ (2,0),\ C\ (0,2),\ D\ (2,2)$.

Now consider a point $E\ (P,Q)$ where P and $P-2$ form one twin-prime pair, and Q and $Q-2$ form another twin-prime pair. Adding this single point produces four prime triangles:

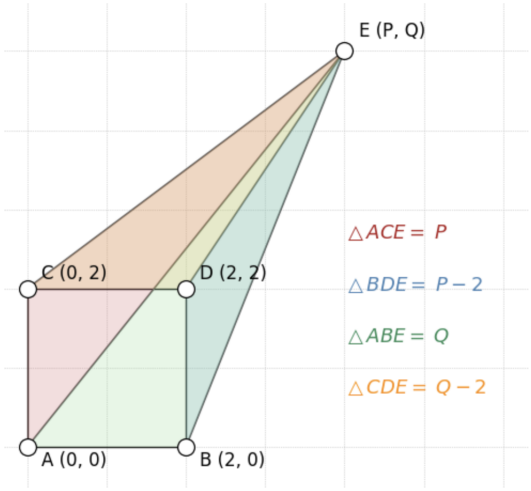

Figure: Four prime triangles from the point (P,Q).

- Triangle $A\ (0,0),\ C\ (0,2),\ E\ (P,Q)$ has area P.
- Triangle $B\ (2,0),\ D\ (2,2),\ E\ (P,Q)$ has area $P-2$.
- Triangle $A\ (0,0),\ B\ (2,0),\ E\ (P,Q)$ has area Q.
- Triangle $C\ (0,2),\ D\ (2,2),\ E\ (P,Q)$ has area $Q-2$.

Since P, $P-2$, Q, and $Q-2$ are all prime, one added point yields four distinct prime areas. Therefore, this construction uses only $\lceil \frac{N}{4} \rceil + 4$ lattice points, which is within the allowed limit of $\lceil \frac{N}{4} \rceil + 6$.

Bonus: For $N \geq 400$, try to solve this using $\lceil \frac{N}{7} \rceil$.

Problem I. Peak Reduction

To delete an element p_i, it must satisfy $1 < i < |p|$ and

$$p_i > p_{i-1} \quad \text{and} \quad p_i > p_{i+1},$$

So, a value can be removed only if both of its neighbours are smaller. The value 1 can never be removed (there is no smaller number), and the value 2 also cannot be removed while 1 is present, since it would need two neighbours smaller than 2, but only one such value exists.

The first and last elements can never be removed, because the deletion rule applies only to positions strictly between 1 and n. Therefore, if the permutation can be reduced to length 2, the remaining two values must be the original endpoints, and these must be exactly $\{1, 2\}$.

Conversely, if the endpoints are $\{1, 2\}$, all elements in positions $2 \dots n-1$ are at least 3. The largest remaining element in this middle segment always has both neighbours smaller than it, so it can be deleted. Repeating this eventually removes all middle elements.

Thus, the answer is `YES` if and only if the two endpoints of the initial permutation are 1 and 2 in any order.

Problem J. The Power of the Sun

A central observation is that the factorial function becomes stable after at most 4 updates when all computations are performed modulo 998244353. More precisely, three cases completely describe the behaviour.

First, $0! = 1$, and since $1! = 1$, an element that is 0 transitions to 1 and then remains fixed.

Second, the values 1 and 2 are already fixed points, because $1! = 1$ and $2! = 2$.

Third, if a value x satisfies $x \geq 998244353$, then $x!$ is divisible by 998244353, implying $x! \equiv 0 \pmod{998244353}$; after that, additional updates keep the value at 0.

Consequently, every position in the array can change at most 4 times before reaching a value that will never change again.

Handling Range Updates via Active Index Tracking:

We maintain a set of "active" indices, i.e. indices whose current values are not yet stable. During a type 1 operation on a range $[l, r]$, we use `lower_bound` to locate the first active index at least l, and then iterate through active indices until exceeding r. Whenever an index reaches a stable value (namely 1, 2, or 0 after a large factorial), it is removed from the active set. Since each index can be updated only a bounded number of times before becoming stable, the total number of such updates over the entire input is linear. Each update requires logarithmic time due to operations on the set, leading to an overall time complexity of order $n \log n$ for all type 1 queries combined.

Efficient Range Sum Queries via Fenwick Tree:

For type 2 queries, we maintain a Fenwick tree over the array values, allowing us to obtain range sums in logarithmic time. To support efficient updates, we precompute all factorials up to 10^6 modulo 998244353. The values 10!, 11!, and 12! require special handling, since their factorials and subsequent factorial iterations must be stored explicitly. We hardcode those values during implementation. For all $x \geq 13!$, we have $x! \equiv 0 \pmod{998244353}$, so any factorial operation on an argument of at least 13! immediately yields 0.

Overall Complexity:

Each type 2 query is answered in $O(\log n)$ time using the Fenwick tree, and the combined complexity of all operations over all test cases is bounded by $O((n + q) \log n)$.

Problem K. The Great Withering

1. Understanding the Process

Step 1:

Let's consider the very **last** node removed. It has no edges left, so its removal cost is 0. Now, suppose at any step we remove a node that is **not** a leaf. Removing a non-leaf node splits the tree into two or more separate components. Each of these components must eventually be completely removed. The last node removed from *each* component would have no edges left, resulting in a cost of 0. This means our sequence of removal costs would contain multiple zeros. However, the problem requires the costs to be **strictly decreasing** ($s_1 > s_2 > \cdots > s_n$). Since edge weights are positive, we can only have exactly one zero at the very end (s_n).

So we are never allowed to split the tree. We must strictly remove leaf nodes one by one until only one node (the root) remains. That means **we must always remove leaves**.

Step 2:

Since we strictly remove leaves, when we pick a node u to remove, it is connected to exactly one remaining neighbor (its parent p). Thus, the removal cost s for node u is simply the weight of the edge (u, p).

So **costs (each s_i) correspond to edge weights**. So as all elements are strinctly decreasing, the edge weights must also be **distinct**.

Step 3:

We must remove a child u before we can remove its parent p. Let $w(u, p)$ be the weight of the edge between u and p, and $w(p, g)$ be the weight between p and its parent g.

- When we remove u, the cost is $w(u, p)$.
- Later, when we remove p, the cost is $w(p, g)$.

Because the costs must be strictly decreasing ($s_{first} > s_{later}$), it implies:

$$w(u, p) > w(p, g)$$

This must hold for every node in the tree.

Edge weights must **strictly decrease** along any path from a leaf towards the root. The edges closest to the root must be the smallest.

Now, when this is satisfied, we can always remove leaves in order of decreasing edge weights to achieve the desired removal cost sequence.

So **the edge weights form a heap like structure. Basically the weight of a parent edge is always smaller than the weights of its children edges**.

2. Counting Valid Assignments

The Formula for a Fixed Root

If we fix a specific node r as the root, the number of ways to assign $n-1$ distinct weights such that they decrease towards r is:

$$\text{Ways}(r) = (n-1)! \times \prod_{v \neq r} \frac{1}{\text{size}_r(v)}$$

where $\text{size}_r(v)$ is the size of the subtree at v when the tree is rooted at r.

Check this out for more details: `https://codeforces.com/blog/entry/75627`

Handling Unknown Roots

We don't know which node is the root, but we know the structure requires weights to decrease towards it. This implies the edge with the **global minimum weight** must be connected to the root. Since an edge connects two nodes, say u and v, both u and v are valid candidates for the root of that specific assignment. If we sum Ways(r) for all nodes r, we effectively count every valid assignment twice (once for each endpoint of the minimum edge).

$$\text{Total Permutations} = \frac{1}{2}\sum_{r=1}^{n} \text{Ways}(r)$$

Choosing the Weights

We need to choose $n-1$ distinct weights from the range $[1, m]$ (summed over all m from 1 to k). The number of ways to choose $n-1$ distinct values from $[1, m]$ is $\binom{m}{n-1}$. So the total is:

$$\sum_{m=1}^{k} \binom{m}{n-1} = \binom{k+1}{n}$$

by the Hockey-stick identity. Check this out for more details: https://en.wikipedia.org/wiki/Hockey-stick_identity

3. Final Algorithm

The final answer is:

$$\binom{k+1}{n} \times \frac{1}{2}\sum_{r=1}^{n} \left((n-1)! \prod_{v \neq r} \frac{1}{\text{size}_r(v)} \right)$$

To compute the sum efficiently:

1. **Base Case:** Compute the product term for node 1 using a standard DFS ($O(n)$).
2. **Re-rooting:** Use a second DFS to move the root to adjacent nodes. When moving the root from u to neighbor v, the subtree sizes only change for u and v. We can update the product in $O(1)$ time using modular inverse.
3. **Total Time Complexity:** $O(n)$.