Important Links

- Main Contest Link: https://toph.co/c/duet-inter-university-iupc-2025
- Practice Link: https://codeforces.com/gym/105884
- Onsite Teams: https://therealbcs.com/team-ratings?contest_id=duet_iupc_2025

Contest Analysis

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. Pair Pressure	Yeamin Kaiser	Yeamin Kaiser	Tasmeem Reza
B. The Last Bit of Us	Shahjalal Shohag	Sayeef Mahmud	Kawsar Hossain
C. Triangle Trap	Shahjalal Shohag	Kawsar Hossain	Al-Shahriar Tonmoy
D. An Interesting Problem	Shahjalal Shohag	Rudro Debnath	MD Irfanur Rahman Rafio
E. Polynomial K Paths	Tasmeem Reza	Tasmeem Reza	Sabbir Rahman Abir
F. Distinct of Distincts	Shahjalal Shohag	Rudro Debnath	Al-Shahriar Tonmoy
G. To Infinity and Beyond	Sabbir Rahman Abir	Sabbir Rahman Abir	Yeamin Kaiser
H. Litmus Test	MD Irfanur Rahman Rafio	MD Irfanur Rahman Rafio	Rudro Debnath
I. XOR This OR That	Shahjalal Shohag	Al-Shahriar Tonmoy	Sachin Deb
J. LCM Factorization	Shahjalal Shohag, Jubayer Nirjhor	Shahjalal Shohag	Sayeef Mahmud

We also tried to provide problems with multiple variations and with a balanced difficulty. 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
H. Litmus Test	Div2A	130	127	Adhoc
D. An Interesting Problem	Div2A	100	126	Strings, Adhoc
F. Distinct of Distincts	Div2B	80	114	Constructive
B. The Last Bit of Us	Div2C	50	87	Trees, Greedy
I. XOR This OR That	Div2C	40	15	Bits, Pigeonhole, Bitmasks
J. LCM Factorization	Div2D	30	60	Number Theory, Combinatorics
C. Triangle Trap	Div2D	20	24	Geometry, Convex Hull
G. To Infinity and Beyond	Div2E	7	4	Trees, Combinatorics, Math
A. Pair Pressure	Div2E	2	3	DP
E. Polynomial K Paths	Div2E	2	0	Graphs, Flows

Solutions to All Problems

https://github.com/ShahjalalShohag/bcs-contest-resources/tree/main/duet_iupc_2025

Tutorials

Problem A. Pair Pressure

Imagine you have $2n$ empty slots for an array: _ _ _ ... _. You fill them one by one. When you decide what to put in the next empty slot, you're essentially choosing to place either:

- The **first occurrence** of some number (let's call this X_1).

- The **second occurrence** of some number (let's call this X_2) whose first occurrence X_1 has already been placed.

The **score** comes from specific instances where placing an X_2 immediately matches an X_1 that was suitably active. An **"active"** first occurrence X_1 refers to an X_1 that make a scoring pair (X_1, X_2) if its corresponding X_2 is chosen as the very next operation in the construction of our array.

Let's define a DP state based on the construction of these arrays by deciding the type of element to place (a first occurrence X_1 or a second occurrence X_2).

Our DP function, say `solve(total_open, scorable_open, total_close)`, will return a pair: $\{\text{ways}, \text{total\_score\_sum}\}$.

DP State Definition

- `total_open`: The number of distinct integers whose first occurrence (X_1) has been placed.
- `scorable_open`: The number of distinct integers x for which X_1 has been placed, X_2 has not, and X_1 is part of a "current active chain."If X_2 for one of these integers is placed *immediately next*, it forms a pair (X_1, X_2) that contributes $+1$ to the score being built for the current array structure.
- `total_close`: The number of distinct integers whose second occurrence (X_2) has been placed.

The DP computes:

- `ways` — the number of ways to complete an array from the current state,
- `total_score_sum` — the sum of all scores for these `ways`.

Base Case

- If `total_open == n`, `total_close == n`, and `scorable_open == 0`: all n integers have had both their first and second occurrences placed, and there's no active scorable chain. This signifies one complete structural arrangement. It contributes 1 way to be in this state and 0 *additional* score at this final step. So, return $\{1, 0\}$.

Transitions

At each step, we consider three types of operations. All calculations are modulo M.

1. **Place a first occurrence (X_1) — "Open a new number":**
   - Possible if `total_open < n`.
   - Number of choices: `remaining_choices` $= n -$ `total_open`.
   - Recursively call: `res = solve(total_open + 1, scorable_open + 1, total_close)`.
   - Update current state:

$$\texttt{ways} = (\texttt{ways} + \texttt{remaining\_choices} \times \texttt{res.first}) \bmod M$$
$$\texttt{total\_score\_sum} = (\texttt{total\_score\_sum} + \texttt{remaining\_choices} \times \texttt{res.second}) \bmod M$$

2. **Place a second occurrence (X_2) and Score — "Close a scorable X_1":**
   - Possible if `scorable_open > 0` and `total_close < n`.
   - Number of choices: `scorable_choices = scorable_open`.

- Adds +1 to the score for each way.
- Recursively call: `res = solve(total_open, 0, total_close + 1)`.
- Update current state:

$$\texttt{ways} = (\texttt{ways} + \texttt{scorable\_choices} \times \texttt{res.first}) \bmod M$$
$$\texttt{total\_score\_sum} = (\texttt{total\_score\_sum} + \texttt{scorable\_choices} \times (\texttt{res.second} + \texttt{res.first})) \bmod M$$

3. **Place a second occurrence (X_2) without immediate scoring — "Close a non-scorable open X_1":**
   - Number of such integers:

$$\texttt{non\_scorable\_choices} = (\texttt{total\_open} - \texttt{total\_close}) - \texttt{scorable\_open}$$

   - Possible if `non_scorable_choices > 0` and `total_close < n`.
   - Recursively call: `res = solve(total_open, scorable_open, total_close + 1)`.
   - Update current state:

$$\texttt{ways} = (\texttt{ways} + \texttt{non\_scorable\_choices} \times \texttt{res.first}) \bmod M$$
$$\texttt{total\_score\_sum} = (\texttt{total\_score\_sum} + \texttt{non\_scorable\_choices} \times \texttt{res.second}) \bmod M$$

The initial call is `solve(0, 0, 0)`. The `total_score_sum` (the second element of the pair returned) is the answer for the test case.

Complexity

- Time Complexity: There are $\mathcal{O}(N^3)$ states. Each state computation involves a constant number of recursive calls and $\mathcal{O}(1)$ work. So, the total time complexity is $\mathcal{O}(N^3)$.
- Space Complexity: $\mathcal{O}(N^3)$ for the memoization table.

Problem B. The Last Bit of Us

Let C_1 be the count of nodes i where $A_i = 1$. Consider the effect of applying the operation on an edge (u, v):

- If $A_u = A_v$, both values flip. If they were 0, C_1 increases by 2. If they were 1, C_1 decreases by 2.
- If $A_u \neq A_v$, one becomes 0 and the other becomes 1. C_1 remains unchanged.

In all cases, the change in C_1 is an even number ($\Delta C_1 \in \{-2, 0, 2\}$). Therefore, the parity of C_1 is an invariant; it never changes.

The goal is to reach a state where all $A_i = 0$, meaning $C_1 = 0$. Since 0 is even, if the initial count C_1 is odd, it's impossible to reach the target state. In this case, the answer is -1.

If the initial C_1 is even, a solution always exists, and the minimum number of operations can be found using a greedy approach.

Algorithm:

1. Root the tree arbitrarily (e.g., at node 1).
2. Perform a traversal (e.g., DFS) to process nodes in a bottom-up manner (children before parent).
3. Keep track of the total operations performed, initialized to 0.

4. For each node u being processed (after all its children): If the current value $A_u = 1$, perform the operation on the edge connecting u to its parent, say p. This flips both A_u (to 0) and A_p. Increment the operation count. (If u is the root and $A_u = 1$, this step is skipped — this situation shouldn't happen if the initial C_1 is even, due to the invariant).

Correctness and Optimality: When processing node u, all nodes in its subtree (except u itself) have already been processed and their values finalized to 0. If $A_u = 1$ at this point, the *only* way to make $A_u = 0$ without affecting its already-processed children is to flip the edge $(u, \text{parent}(u))$. Therefore, if $A_u = 1$, this operation is necessary. The algorithm performs an operation only when it's necessary for the node currently being processed. Since each operation is tied to a necessary flip for a specific node u at the time it's processed from the bottom up, the total count of operations is the minimum required. The change to $A_{\text{parent}(u)}$ is handled when the parent node is processed later in the traversal.

Because the initial C_1 is even, the root node's value will end up as 0 after the process completes.

We can use DFS or BFS to implement the algorithm.

Problem C. Triangle Trap

First, compute the convex hull of the given n points $p_1, \ldots, p_n$. This can be done in $O(n \log n)$ time using standard algorithms like Graham Scan or Monotone Chain. Let H be the set of indices of the points lying on the convex hull.

Iterate through the point indices $s = 1, 2, \ldots, n$. Check if index s belongs to the set H. If $s \notin H$, then the point p_s must lie strictly inside the convex hull (since no three points are collinear). According to the problem statement, we need the solution with the smallest index i. Therefore, the first index s encountered during this iteration such that $s \notin H$ is our target index i. If all points lie on the convex hull (i.e., H contains all indices from 1 to n), then no point lies strictly inside any triangle formed by other points. In this case, no solution exists, and we should output -1.

Once the smallest index i (such that p_i is strictly inside the hull) is found, we need to find three distinct hull vertices p_j, p_k, p_l (where $j, k, l \in H$ and $j, k, l \neq i$) such that p_i lies strictly inside $\triangle p_j p_k p_l$. Since p_i is strictly inside the convex hull, such a triangle formed by hull vertices must exist. A simple way to find one such triangle is:

1. Pick an arbitrary hull vertex index $j \in H$.
2. Iterate through all edges (p_k, p_l) of the convex hull polygon, where $k, l \in H$ are indices of adjacent vertices on the hull.
3. For each triangle $\triangle p_j p_k p_l$ formed, perform a point-in-triangle test to check if p_i lies strictly inside it. This test can be done efficiently in $O(1)$ time, for example, by checking if p_i lies on the same side of all three oriented edges (p_j, p_k), (p_k, p_l), and (p_l, p_j).
4. Since we are guaranteed that p_i lies inside the hull, this procedure will eventually find a valid triangle $\triangle p_j p_k p_l$ containing p_i. Output the first valid set of indices (i, j, k, l) found.

Overall complexity is $O(n \log n)$.

Problem D. An Interesting Problem

Claim: A string is interesting if and only if all its characters are unique.

Proof:

- If all characters in s are unique: For any substring t, we have $f(t) = 1$ (each substring appears exactly once). Therefore, $|t| \bmod f(t) = |t| \bmod 1 = 0$ is always true. Hence, the string is interesting.

- If any character appears multiple times: Consider a substring t containing just one character (that appears multiple times). Then $|t| = 1$ and $f(t) > 1$, which gives $|t| \bmod f(t) = 1 \bmod f(t) \neq 0$. This violates the interesting property. For example, if $s =$ `aba`, then $t =$ `a` is a substring that appears twice, and $1 \bmod 2 = 1 \neq 0$.

So just check if all characters in the string are unique or not.

Problem E. Polynomial K Paths

We will solve this problem by using min cost max flow. First of all, notice that the function $g(x) = xf(x)$ is convex.

$$\frac{\mathrm{d}}{\mathrm{dx}}g(x) = \sum_{i=0}^{d}(i+1)c_i x^i \geq 0$$

So $g(x) - g(x-1)$ is a non-decreasing function.

Now we can construct the following flow network $N = (V_N, E_N)$ for our problem. The network will consist of all the nodes of nodes from G as well as a source node s and sink node t. For each edge $u \to v$ in G, add k duplicate edges from $u \to v$ in N, with the i $(i = 1, 2, \ldots, k)$ th edge having 1 capacity and $g(i) - g(i-1)$ cost. Additionally, add an edge from $s \to 1$ with capacity k cost 0. Similarly add an edge $n \to t$ with capacity k cost 0. Now run min cost max flow algorithm from s to t. The minimum cost is the desired answer.

Now let's prove this. First of all, the max flow amount must be equal to exactly k. It cannot be more than k since $s \to 1$ has capacity k and it cannot be less than k since there exists a path from $1 \to n$, so we could simply push k flow along that path.

Now let's apply flow decomposition theorem. We know the maximum flow could be decomposed into k paths. If an edge $u \to v$ is used X times by those k paths, then the X edges with the lowest costs will be used. Those costs are $g(1)-g(0)$, $g(2)-g(1)$, $\ldots$, $g(X)-g(X-1)$ since $g(x)-g(x-1)$ is a non-decreasing function. So the total cost contribution of the edge $u \to v$ is exactly

$$\sum_{i=1}^{X}(g(i) - g(i-1)) = g(X) - g(0) = g(X) = Xf(X)$$

which is exactly the cost incurred by edge $u \to v$ in the original graph. So computing the min cost max flow algorithm on network N gives us the correct answer. The MCMF complexity turns out to be $O(E_N \cdot \text{flow} \cdot \log V_N) = O(k^2 m \log N)$

As a final note, if your code template doesn't support multi-edges, you can use dummy nodes to get around this problem. It will increase the number of vertices to $O(km)$, but the overall complexity stays similar.

Problem F. Distinct of Distincts

In this problem, we need to fill an $n \times m$ grid with integers to maximize the number of distinct counts in set S, where S contains the counts of distinct elements in each row and column.

The maximum possible number of distinct counts in set S is:

$$\min(2 \cdot \min(n, m), \max(n, m))$$

To simplify our analysis, we'll assume $n \leq m$ (rows $\leq$ columns). If this is not the case $(n > m)$, simply swap the roles of rows and columns in the solution approach.

Upper Bound Analysis:

First, let's establish the upper bound on distinct counts (assuming $n \leq m$):

- Each column can have at most n distinct elements (one per row), so $c_i \leq n$ for all i.
- Each row can have at most m distinct elements (one per column), so $r_i \leq m$ for all i. But we have only n rows. So, we can only have at most n distinct counts over all rows.
- Therefore, the maximum possible size of set S is $n + n = 2n$.
- But maximum of all r_i and c_i is bounded by m. So, we can only have at most m distinct counts in set S.
- Therefore, the maximum possible size of set S is $\min(2n, m)$.

Example Construction Strategy:

For $n \leq m$ (rows $\leq$ columns), our strategy is to:

1. Create as many different column counts as possible (limited by n)
2. Use the remaining freedom to create different row counts

Step 1: Create columns with $1, 2, \ldots, n$ distinct elements.

For example, with $n = 3, m = 5$:

1	1	1	?	?
1	1	2	?	?
1	2	3	?	?

This gives column counts $c_1 = 1, c_2 = 2, c_3 = 3$.

Step 2: Fill the remaining columns to maximize row distinctness:

1	1	1	4	5
1	1	2	4	5
1	2	3	4	5

This gives row counts $r_1 = 3, r_2 = 4, r_3 = 5$.

In set S, we have $\{1, 2, 3, 4, 5\}$ which is 5 distinct values. This matches our formula: $\min(2 \cdot 3, 5) = \min(6, 5) = 5$.

Similarly, you can solve for other grids. Check the author's solution for better understanding.

Problem G. To Infinity and Beyond

Solution 1: Strict Inequality and Subtree Sizes

Instead of working with the condition $A_i \leq A_{a_i}$, consider the strict condition $A_i < A_{a_i}$. Let this count be $\text{count}'_a(x)$. Then:

$$\lim_{x \to \infty} \frac{\text{count}_a(x)}{\text{count}_b(x)} = \lim_{x \to \infty} \frac{\text{count}'_a(x)}{\text{count}'_b(x)}$$

There is a formal proof which is bit large, but informally, as x gets large, assignments with duplicate numbers become very small compared to the full number of assignments

We compute $count'_a(x)$ in following way (similar for b):

1. Choose n distinct values from $\{1, \ldots, x\}$ to assign to the n nodes. This is $\binom{x}{n}$ ways.

2. Assign the chosen n values to nodes such that $A_i < A_{a_i}$ for all $i > 1$.

This second step is a standard combinatorics problem. We can view the array a as defining a rooted tree, where each $i > 1$ has parent a_i, and 1 is the root.

In such a tree, the number of permutations of $\{1, \ldots, n\}$ that satisfy the constraint $A_i < A_{\text{parent}(i)}$ is:

$$\text{ways} = \frac{n!}{\prod_{i=1}^{n} s_i}$$

where s_i is the size of the subtree rooted at node i. There is also a solution that uses combinations to assign number at root and divide numbers to children which ultimately gives the same formula.

You can also compute it using dynamic programming on the tree. Check out this blog to understand the above formula better: https://codeforces.com/blog/entry/75627

Thus,

$$\text{count}'_a(x) = \binom{x}{n} \cdot \frac{n!}{\prod_i s_i}$$

Similarly for b. In the ratio, $\binom{x}{n} \cdot n!$ cancels out. So:

$$r = \frac{\prod_i s_i^{(b)}}{\prod_i s_i^{(a)}}$$

Solution 2: Leading Coefficient of the Polynomial

We note that $\text{count}_a(x)$ and $\text{count}_b(x)$ are polynomials in x of degree n. As $x \to \infty$, the highest degree term dominates, so we only need the ratio of leading coefficients.

We compute the leading coefficient recursively using a DP over the tree defined by a or b.

Let $\text{dp}[i]$ denote the leading coefficient of the number of valid assignments in the subtree rooted at i. Suppose node i has children $c_1, \ldots, c_k$ with subtree sizes s_{c_j}. Then:

$$\text{dp}[i] = \prod_{j=1}^{k} \left(\frac{\text{dp}[c_j]}{s_{c_j}} \right)$$

The intuition is that the number of ways to assign values 1 to y in the subtree is a degree-d polynomial with leading coefficent p (where d is the subtree size), and summing such polynomials over 1 to y produces another polynomial where leading coefficient is $\frac{p}{d}$. For a child c with subtree size s, this contributes a factor of $\frac{dp[c]}{s}$.

So:

$$\text{dp}[i] = \frac{1}{\prod_j s_{c_j}} \cdot \prod_j \text{dp}[c_j]$$

Using this dp, we get leading coefficient at root $dp[1]$. Final answer is ratio of that for the two arrays.

Problem H. Litmus Test

- To verify a chemical is **acid**, it is necessary and sufficient that we use a blue litmus paper and observe it turn red.
- To verify a chemical is **base**, it is necessary and sufficient that we use a red litmus paper and observe it turn blue.
- To verify a chemical is **salt**, it is necessary and sufficient that we use both a red and a blue litmus paper and observe no change in either.

Let a and b be the number of acids and bases, respectively.

Consider the following cases:

- $a > 0$ **and** $b > 0$**:**
  Since we have both acids and bases, we can use acids to make any paper red and bases to make any paper blue whenever we want. So we need one litmus paper to start, and then we can always reuse that by changing its color as needed.
  Answer $= 1$
- $a = 0$ **and** $b > 0$**:**
  Once a red paper turns blue (after testing a base), we cannot make it red again. So each base requires a separate red paper.
  For salts, we don't need any extra paper. We can use one red paper before testing any base and one blue paper obtained by using a red paper on a base to verify all salts.
  Answer $= b$
- $b = 0$ **and** $a > 0$**:**
  Similarly, each acid requires a separate blue paper.
  For salts, we don't need any extra paper. We can use one blue paper before testing any acid and one red paper obtained by using a blue paper on an acid to verify all salts.
  Answer $= a$
- $a = 0$ **and** $b = 0$**:**
  All chemicals are salts. We need one red and one blue paper to confirm this. Neither will change color, and we can reuse those 2 papers on all chemicals.
  Answer $= 2$

Problem I. XOR This OR That

The core task is to partition the given array a into two non-empty subsets, S_1 and S_2, such that $(\bigoplus_{x \in S_1} x) \times (\bigvee_{y \in S_2} y)$ is minimized. Here, $\bigoplus$ denotes the bitwise XOR operation, and $\bigvee$ denotes the bitwise OR operation.

We observe that each element $a_i \leq 10^6$. Since $2^{19} < 10^6 < 2^{20}$, each element fits within 20 bits. This implies that the bitwise XOR sum of any subset of elements will also fit within 20 bits, meaning there are at most 2^{20} possible distinct XOR sums.

Case 1: $n > 20$

The number of non-empty subsets of the array a is $2^n - 1$. Since $n > 20$, we have $2^n - 1 > 2^{20}$. By the pigeonhole principle, since there are more non-empty subsets than possible distinct XOR sums (at most 2^{20}), there must exist at least two distinct non-empty subsets, say A and B, such that their elements have the same XOR sum: $\bigoplus_{x \in A} x = \bigoplus_{y \in B} y$.

Consider the set S_1 formed by the elements that are in either A or B, but not both (symmetric difference $A \Delta B$). Since A and B are distinct, S_1 is non-empty. The XOR sum of elements in S_1 is:

$$\bigoplus_{z \in S_1} z = \left(\bigoplus_{x \in A} x \right) \oplus \left(\bigoplus_{y \in B} y \right) = 0$$

(This works because elements common to both A and B appear twice in the combined XOR, canceling themselves out).

We have found a non-empty subset S_1 such that $\bigoplus_{z \in S_1} z = 0$. Let S_2 be the set of all elements in a that are not in S_1. Since $n \geq 2$ and S_1 is non-empty, S_1 cannot contain all elements (as A and B are subsets of a), guaranteeing that S_2 is also non-empty. The value for this partition (S_1, S_2) is

$(\bigoplus_{z\in S_1} z) \times (\bigvee_{w\in S_2} w) = 0 \times (\bigvee_{w\in S_2} w) = 0$. Since the product cannot be negative, 0 is the minimum possible value. Therefore, if $n > 20$, the answer is always 0.

Case 2: $n \leq 20$

When n is small, we can iterate through all possible ways to partition the n elements into two sets, S_1 and S_2. There are 2^n total ways to assign each element to either S_1 or S_2. We must exclude the two cases where either S_1 or S_2 is empty, leaving $2^n - 2$ valid partitions.

For each valid partition (S_1, S_2):

- Calculate the bitwise XOR sum $X = \bigoplus_{x\in S_1} x$.
- Calculate the bitwise OR sum $Y = \bigvee_{y\in S_2} y$.
- Keep track of the minimum value of $X \times Y$ encountered across all valid partitions.

This can be implemented, for example, using recursion or iteration with bitmasking in $O(n \cdot 2^n)$ or $O(2^n)$ time complexity.

Problem J. LCM Factorization

A prime p contributes to $f(\text{LCM}(\dots))$ if and only if p divides the LCM, which means p must divide at least one a_i in the subsequence.

Instead of iterating through subsequences, we can sum the contributions of each prime p. The total sum S can be written as:

$$S = \sum_{p \text{ prime}} p \cdot (\text{Number of } k\text{-subsequences whose LCM is divisible by } p)$$

Fix a prime p. We need to count the number of k-subsequences $\{a_{i_1}, \dots, a_{i_k}\}$ where at least one a_{i_j} is divisible by p.

We use complementary counting. Let N_p be the count of numbers in the input array a that are divisible by p. There are $n - N_p$ numbers in a not divisible by p.

Total k-subsequences $= \binom{n}{k}$ and k-subsequences with *no* element divisible by $p = \binom{n-N_p}{k}$.

So, the number of k-subsequences with *at least one* element divisible by p is $C_p = \binom{n}{k} - \binom{n-N_p}{k}$.

Since $a_i \leq n$, we only consider primes $p \leq n$. The answer is:

$$\sum_{\substack{p \text{ prime} \\ p\leq n}} p \cdot \left(\binom{n}{k} - \binom{n - N_p}{k}\right)$$

So use sieve to compute the number of elements divisible by each prime $p \leq n$ and precompute factorials and modular inverses up to n to compute $\binom{n}{r}$ efficiently. Then just iterate through all primes $p \leq n$ and sum up the contributions.