SMU ICPC Selection Contest Editorial(v1.0)

18 July 2026

SMU Competitive Programming

Official Editorial

Problem and Editorial Author: Jonathan Teo (jonteo_2001)

CF Gym Preparation: Tai Tze Kin (teekaytai)

Testers: Pham Tuan Thanh (Floating_Duck)
Tai Tze Kin (teekaytai)
Yu Peng Ng (feeder1)
Tan Tiong Guan (tiongg)

Problem A. Arrival Signs

Difficulty Estimate	CF 1000
Topics / Techniques	Implementation, Modular Arithmetic
Solved by	16 participants
First Solve	Hai Dang `danglayloi1`, Pasit`hexahydride` (6 minutes)

Consider the two cases of the arrival time at platform j

- If a commuter arrives at a time y that is a multiple t_j ($y \bmod t_j = 0$), then the commuter boards the train immediately.
- If a commuter arrives at a time y that is not a multiple of t_j ($y \bmod t_j \neq 0$), then the commuter must wait additionally $t_j - (y \bmod t_j)$ after reaching the platform.

Solution

Let y denote the arrival time for a query (x, i, j), which can be computed using $y = x + d_{i,j}$. Then the answer is

$$\text{answer} = \begin{cases} d_{i,j}, & \text{if } y \bmod t_j = 0 \\ d_{i,j} + t_j - (y \bmod t_j), & \text{otherwise} \end{cases} \tag{1}$$

Complexity

Time Complexity: $\mathcal{O}(nm + q)$, where m is the number of signs, n is the number of platforms, and q number of queries

Problem B. Billy Loves Zeros

Difficulty Estimate	CF 1200
Topics / Techniques	Greedy
Solved by	14 participants
First Solve	Hai Dang `danglayloi1` (11 minutes)

Observation 1. *Consider a permutation* $1 \cdots n$ *right-shifted, and we get at most two increasing subarrays where each position has a difference of* 1.

Consider a permutation $a = [1, 2, \cdots, n]$ is right-shifted by k positions. We get the permutation

$$b = [\underbrace{n-k+1, n-k+2, \cdots, n}_{\text{increasing subarray}}, \underbrace{1, 2, \cdots, n-k}_{\text{increasing subarray}}]$$

When either subarray has length larger than 1, we get elements that increase by exactly 1.

Observation 2. *Let an increasing subarray with differences of* 1 *at adjacent positions be* $a = [x, x+1, \cdots, x+y-1, x+y]$*, for some* $y \geq 0$*. a can be made into* $[0, 0, \cdots, 0]$ *using exactly* $m - x$ *operations, and this is the minimum for a.*

Proof. Let $f(l, r)$ denote that we apply the operation in the range $[l, r]$. We can do $m - x$ operations like this:

- We perform the operation $f(1, y+1)$ a total of $m - (x+y)$ times. Then $a_{y+1} = 0 \bmod m$. Note the array is

$$\begin{aligned} a' &= [x + m - (x+y), x + m - (x+y) + 1, \cdots, x + y - 1 + m - (x+y), 0] \\ &\Rightarrow [m - y, m - y + 1, \cdots, m - 1, 0] \end{aligned} \tag{2}$$

- We perform $f(1, y)$ once. Then position $a_y = 0 \bmod m$
- We perform $f(1, y-1)$ once. Then position $a_{y-1} = 0 \bmod m$

... and so on. In total we perform exactly $m - (x+y) + y = m - x$ operations.

Why is this the minimum? The array a has a minimum x, and it needs at least $m - x$ operations for the minimum x to become 0. □

Observation 3. *Solve for two increasing subarrays with differences of* 1 *at adjacent positions.*

Proof.

$$b = [\underbrace{n-k+1, n-k+2, \cdots, n}_{\text{SA1}}, \underbrace{1, 2, \cdots, n-k}_{\text{SA2}}]$$

If we solve both in isolation, SA1 would require $m - (n - k + 1)$ operations, and SA2 requires $m - 1$ operations. But notice, since $n < m$, we can apply $f(1, n)$ until the position of n becomes 0. In doing so, we *save* $m - n$ operations.

Therefore, the minimum number of operations for two increasing subarrays with differences of 1 at adjacent positions is

$$m - (n - k + 1) + (m - 1) - (m - n) = m + k - 2 \tag{3}$$

□

Solution

Therefore, the answer can be summarised by a formula, but we must handle the case when the array is not rotated at all:

$$\text{answer} = \begin{cases} m-1, & \text{if } k = 0 \\ m+k-2, & \text{otherwise} \end{cases} \tag{4}$$

Complexity

$\mathcal{O}(1)$ per test case.

Problem C. Can you BELIEVE it?

Difficulty Estimate	CF 1400
Topics / Techniques	Constructive Algorithms, Residue Classes, Brute Force
Solved by	11 participants
First Solve	Yvonnie `coutopenbeta` (64 minutes)

A blind brute force would take too long, but $n = 52$ is small enough to hardcode an answer. We can use some principles to guide our hardcoding.

Observation 1. *Handwritten letters at intervals of* 7 *must be equal.*

Proof. Across every interval of length 7, the multiset of handwritten letters must be the same. For the multisets of letters to be the same, the handwritten letter on card i must be equal to the handwritten letter on card $i + 7$. □

Observation 2. *When sliding the window from* $[i, i+6]$ *to* $[i+1, i+7]$*, the new card has the same letter as the removed card, so its rank must fit the same relative sorted position among the new 7-card window.*

Solution

This suggests the following construction for $n = 52$.

First, fix the face-down letters to repeat the word `BELIEVE` (Observation 1).

? ?
B E L I E V E B E L I E V E B E L I E V E B E L I E V E B E L I E V E B E L I E V E B E L I E V E B E L

Now, consider the seven positions in each position of BELIEVE as levels of priority: highest priority is the letter `B`, second highest priority is the first letter `E`, and so on. (Note that a letter E later in believe must have a lower priority than an earlier letter).

We assign card numbers in descending order of priority. Among all cards with maximum priority, assign the smallest available card numbers from left to right.

Equivalently, order all positions by

$$(\text{descending priority}, \text{increasing position index})$$

and assign the sequence

$$1, 1, 1, 1, 2, 2, 2, 2, \cdots, 13, 13, 13, 13$$

in this order.

Correctness

- Each card number from 1 to 13 is used exactly 4 times, so the deck constraint is satisfied.
- This also has property that every window of length 7 will spell `BELIEVE` in descending order of priority.
- Consider two consecutive priorities $r + 1$ and r. The card with priority $r + 1$ is either from the same copy of `BELIEVE` as the card with priority r, or from the previous copy. In our construction order, this means the priority-$r + 1$ cards appears at least 7 positions before the priority r card. Since each face-up number is assigned to 4 consecutive positions in the construction order, the priority r card must have a strictly larger face-up number than the priority $r + 1$ card. Hence, in every window, the face-up numbers strictly increase as priority decrease, so sorting the cards by face-up number gives us the letters `BELIEVE`.

This is a few steps of the construction. Assign the smallest number available to every first character of BELIEVE from left to right

1	?	?	?	?	?	?	1	?	?	?	?	?	?	1	?	?	?	?	?	?	1	?	?	?	?	?	?	2	?	?	?	?	?	?	2	?	?	?	?	?	?	2	?	?
B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L

1	3	?	?	?	?	?	1	3	?	?	?	?	?	1	3	?	?	?	?	?	1	3	?	?	?	?	?	2	4	?	?	?	?	?	2	4	?	?	?	?	?	2	4	?	?	?	?	?	2	4	?
B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L

and so on until we complete the construction. Final construction for $n = 52$:

1	3	5	7	8	10	12	1	3	5	7	10	12	9	1	3	5	7	9	11	12	1	3	5	7	9	11	13	2	4	6	8	9	11	13	2	4	6	8	10	11	13	2	4	6	8	10	12	13	2	4	6
B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L	I	E	V	E	B	E	L

Problem D. Duel of Ten

Difficulty Estimate	CF 1500
Topics / Techniques	Games, Constructive Algorithms
Solved by	11 participants
First Solve	Ngoc Phuc `stag3` (82 minutes)

The elements of a fall into 3 categories: $a_i \leq -10$, $a_i \in [-9, 9]$, and $a_i \geq 10$. Define three sets:

- A to be the set of elements ≤ -10
- B to be the set of elements $\in [-9, 9]$
- C to be the set of elements ≥ 10

Each operation decreases the size of one set A, B or C, by exactly 1 element. Let $a \in A, b \in B, c \in C$ in the choice of x, y below.

Pair (x, y) chosen	Alice's turn	Bob's turn
a, b	b is removed, $\|B\|$ decreases by 1	a is removed, $\|A\|$ decreases by 1
a, c	c is removed, $\|C\|$ decreases by 1	a is removed, $\|A\|$ decreases by 1
b, c	c is removed, $\|C\|$ decreases by 1	b is removed, $\|B\|$ decreases by 1

Lastly, if (x, y) are chosen from the same set S, then the size of the set $|S|$ will decrease by 1 after any operation. Alice wins if the only element left in a is in set B, while Bob wins if the only element left in a is in set A or C.

Observation 1. *For Bob, any move using an element in set A is disadvantageous.*

This is because regardless if a, b is chosen or a, c is chosen, Bob removes an element from set A (which he needs to win). Therefore, the only move advantageous for Bob is to choose (b, c) where possible.

Observation 2. *For Alice, any move using an element in set A is either disadvantageous, or there exists an equally optimal move that does not involve the element in set A.*

- If a, b is chosen, Alice removes an element from set B, which she needs to win.
- If a, c is chosen, Alice removes an element from set C. This is advantageous for Alice.
  - If $|B| > 0$, there exists an equally optimal move by choosing the pair (b, c), which she needs to win.
  - If $|B| = 0$, Alice will already lose the game eventually, so choosing the pair (a, c) has no impact on the outcome of the game.

Therefore, the only move advantageous for Alice is to choose (b, c) where possible.

Solution

In conclusion, the optimal game play can be summarized as follows:

1. Alice and Bob will repeatedly do operation (b, c), until either $|B| = 0$ or $|C| = 0$.
2. If $|B| = 0$, then Bob must win eventually.
3. If $|B| > 0$ and $|C| = 0$, then consider
   - If $|A| = 0$, then the player who makes the last move wins the game.
   - If $|A| > 0$, then the player who makes the last move *loses* the game. (See (a, b) row in the table, notice that the player who makes the last move will end up losing the game.)

Note that $|B| = 0$ happens if and only if $|B| < |C|$, and $|C| = 0$ happens if and only if $|B| \geq |C|$. So we don't actually need to simulate step 1.

Complexity

$\mathcal{O}(n)$ time, bottle-neck by reading in the array a.

Correctness

This section provides a formal proof for Alice's winning condition.

Theorem. *Assuming both players play optimally, Alice wins if and only if*

$$|B| \geq |C|, \quad |B| > 0$$

and one of the following holds:

- $|A| = 0$ *or*
- $|A| > 0$ *and* n *is odd*

Proof. Denote a state of the game S by a 3-dimensional state: $(|A|, |B|, |C|)$ Since the end state

of the game contains exactly one element, Alice wins if and only if $(|A|, |B|, |C|) = (0, 1, 0)$, and Bob will win if $S = (1, 0, 0)$ or $S = (0, 0, 1)$.
There are three ways to reach Alice's win state $(0, 1, 0)$.

1. On Alice's turn, transition from state $(0, 1, 1)$ to $(0, 1, 0)$ by Alice choosing (b, c)
2. On Bob's turn, transition from state $(1, 1, 0)$ to $(0, 1, 0)$ by Bob choosing (a, b).
3. On Alice or Bob's turn, transition from state $(0, 2, 0)$ to $(0, 1, 0)$ by choosing (b, b).

sufficiency: Suppose $|B| \geq |C|, |B| > 0$ and either $|A| = 0$ or $|A| > 0$ and n is odd. We show that Alice has a winning strategy.
In all cases, $|B| \geq |C|$ and $|B| > 0$ when it is Alice's turn.

Lemma. *Suppose initially* $|B| \geq |C|$ *and* $|B| > 0$ *and its Alice's turn. Alice has a strategy such that* $|B| \geq |C|$ *at the beginning of every one of her turns, regardless of what Bob does.*

Proof. While $|C| > 0$, Alice chooses (b, c). This decreases $|C|$ by 1, so immediately after Alice's move, we have $|B| > |C|$. On Bob's turn, $|B|$ can decrease by at most 1 (when Bob chooses (b, c)). Therefore, after Bob's move, $|B| \geq |C|$.

Thus, by induction, $|B| \geq |C|$ holds at the beginning of every Alice turn. □

Hence, C must be empty before B. Since initially $|B| > 0$, at least one element of B remains when this happens.

We have two cases. When $|A| = 0$, only elements of B remain, so the game eventually ends in $(0, 1, 0)$. When $|A| > 0$, we show that Alice can win when n is odd:

Lemma. *Suppose* $|A| > 0, |B| > 0, |C| = 0$ *and* n *is odd. Alice has a winning strategy.*

Proof. When n is odd, Bob makes the last move of the game (there are $n - 1$ turns, which is even). Alice strategy is to never remove the last element of B:

- If $|A| \geq 2$, choose (a, a).
- Otherwise $|A| = 1$. If the game has not ended and it is Alice's turn, there must be $|B| \geq 2$, so choose (b, b).

Thus, Alice never makes B empty. Bob also cannot remove the final element of B: when $|B| = 1$, he cannot choose (b, b), and choosing (a, b) removes an element of A, not B.

Therefore, B remains non-empty until the end. Since Bob makes the last move, we transition from state $(1, 1, 0)$ to state $(0, 1, 0)$, and Alice wins. □

Necessity: Suppose Alice has a winning strategy. We show that the stated conditions must hold.
We violate each condition independently to show that Alice cannot win the game.

Firstly, if $|B| = 0$, Alice cannot win, since B can never become non-empty. Hence suppose $|B| > 0$.

Lemma. *Suppose initially* $0 < |B| < |C|$ *and its Alice's turn, Bob has a strategy that preserves* $|B| < |C|$ *at the beginning of each Alice turn, regardless of what Alice does, and Alice loses.*

Proof. Suppose $|B| < |C|$ at the beginning of Alice's turn. After Alice's move, $|C|$ can decrease by at most 1. Hence either $|B| < |C|$ still holds or $|B| = |C|$. Bob chooses (b, c), removing one element from B. Therefore, after Bob's move $|B| < |C|$ and it is Alice's turn. □

Consequently, B must be empty before C, and Alice loses.

Lemma. *Suppose* $|A| > 0$ *and* n *is even. Bob has a winning strategy.*

Proof. When n is even, Alice makes the last move of the game. Bob's winning strategy is to never remove the last element of A. Bob never removes the last A. If $A = 1$, Alice cannot remove it either: (a, a) is unavailable, while (a, b) or (a, c) removes the non-A element.

Therefore, A remains non-empty until the end. Since Alice makes the last move, we transition from state $(1, 1, 0)$ to state $(1, 0, 0)$, and Bob wins. □

□

Problem E. Expected Walk Maximization

Difficulty Estimate	CF 1600
Topics / Techniques	Tree Diameter, Probabilities, Constructive Algorithms
Solved by	7 participants
First Solve	Pasit `hexahydride` (55 minutes)

Observation 1. *Our answer will never exceed the weighted diameter of the tree.*

By definition of the diameter, the diameter of the tree is always the longest path in the tree.

Observation 2. *Alice is always able to walk one diameter of T deterministically.*

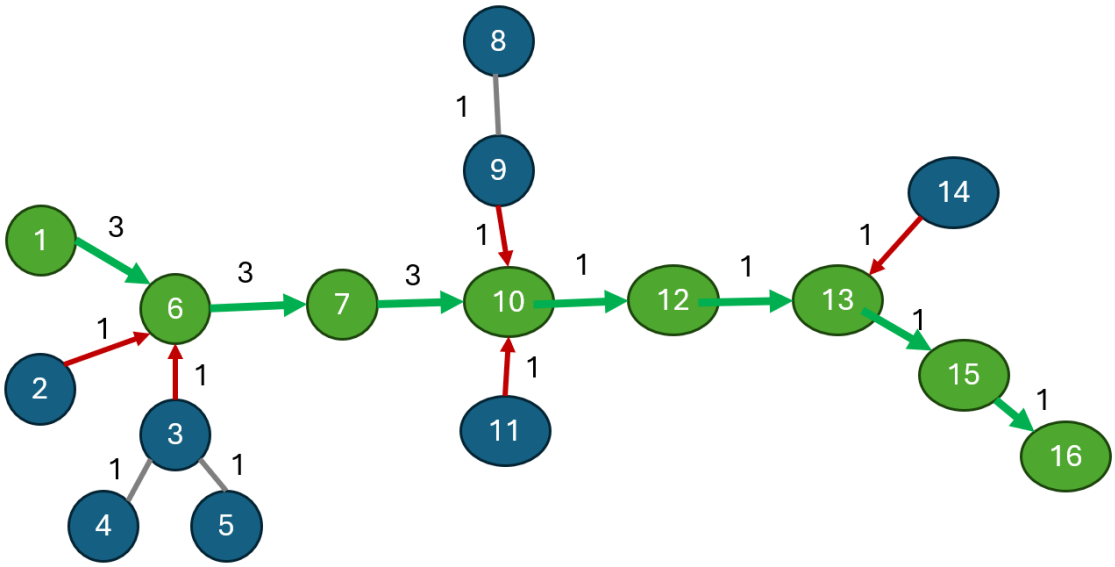

Figure 1: We can always orient edges such that Alice can start walking from one end of the diameter to the other end. In the above figure, the chosen diameter and the edges along the chosen diameter are colored green. Edges that have only one endpoint on the chosen diameter are colored red (oriented toward the diameter), while grey edges denote that any orientation can be chosen for that edge as it does not affect Alice's path.

Proof. Suppose we choose a diameter of the tree, given by the path $s_1 s_2 s_3 s_4 \cdots s_k$ where $1 \leq s_i \leq n$ (all s_i are distinct). Note that adjacent nodes on the diameter must be connected by initially undirected edges in T. Since the diameter is the longest path in a tree, we can orient edges in T' along the path of the chosen diameter that form a path $s_1 \to s_2$, $s_2 \to s_3$, $\cdots$, $s_{k-1} \to s_k$.

For any node i not on the chosen diameter that has an edge with a node s_i on the chosen diameter, we can direct an edge $i \to s_i$ (this prevents Alice from walking from s_i to node i). □

Solution

Calculate D, the weighted diameter of the tree T. The maximum expected walk is deterministically the diameter of T. So output $D \bmod 10^9 + 7$.

Complexity

$\mathcal{O}(n)$ time, where n is the number of nodes.

Problem F. Flawed Prize Redemption

Difficulty Estimate	CF 1600
Topics / Techniques	Knapsack DP, Number Theory, Greedy
Solved by	13 participants
First Solve	Looi Le `looile` (69 minutes)

Observation 1. *For a fixed prize number p_i, we can greedily select the task with the smallest duration.*

Proof. Suppose we have two tasks $a_1 < a_2$ such that $p_1 = p_2$. If Billy can receive a maximum x dollars if he pursues task a_2, he can replace task 2 with task 1 and be left with an additional time unit $a_2 - a_1$ for other tasks, and still receive a maximum x dollars. □

Observation 2. *Let t_i be the minimum task duration possible to obtain a prize i. We can reformulate into a 0-1 knapsack.*

Let $f(i, w)$ denote the maximum prize money possible considering up to the first i items and w units of time. We arrive at the recurrence

$$f(i,w) = \max \begin{cases} f(i-1, w-t_i) + v_i, & \text{if } w - t_i \geq 0 \\ f(i-1, w) \end{cases} \tag{5}$$

and $f(i, 0) = f(0, j) = 0$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$ as a base case.

Observation 3. *Note that $p_i = \lfloor \frac{m}{a_i} \rfloor$, and there are at most $2\lfloor \sqrt{m} \rfloor$ distinct values of $\lfloor \frac{m}{i} \rfloor$.*

We discussed this in class (SMU Week 13 CP Contest), please see the proof here.

We can use this to optimise our Knapsack DP.

Solution

1. Find the minimum possible task duration t_i for each prize number i.
2. Solve using $0-1$ Knapsack DP with each prize as items (there will be at most $2\sqrt{m}$ valid prizes).

Complexity

$\mathcal{O}(n + m\sqrt{m})$ time, for n tasks and m units of time. Step 1 takes $\mathcal{O}(n)$ and step 2 takes $\mathcal{O}(m\sqrt{m})$ time.

Problem G. Google Maps Does Not Work Here

Difficulty Estimate	CF 1800
Topics / Techniques	Minimum Spanning Tree, DP on Tree
Solved by	6 participants
First Solve	Pasit `hexahydride` (82 minutes)

Pitfalls

If you got Wrong Answer, maybe you tried to store all edge weights 2^x mod 998244353 and run Dijkstra. This fails because Dijkstra's Algorithm requires edge weights *before* the modulo operation, not after. For example, the 30-th edge has distance 2^{29} mod 998244353 = 536870912 but the 31-st edge has distance 2^{30} mod 998244353 = 75497471. So running Dijkstra on edge weights after modulo is incorrect.

Observation 1. *The x-th edge is more expensive than traversing all $x-1$ edges that came before it in the edge list.*

Proof. The x edge weight is 2^{x-1}, this corresponds to the binary number `100...0`, with only the $x-1$-th bit set. If a path that traverses all $(x-1)$ edges exists, its combined sum would be $2^{x-2} + 2^{x-3} + \cdots + 2^0$, which is equivalently the binary number `011...1`, this is equal to $2^{x-1} - 1$. □

As such, for any two nodes u, v, if a path from u to v exists in the first k edges, any later edge with position larger than k that also connects (u, v) will never be cheaper.

Observation 2. *Let G be the graph formed using the n vertices and m edges. The shortest path of any pair of nodes $(u, v), u \neq v$ will always traverse edges of the Minimum Spanning Tree of G.*

Proof. Less formal proof: Suppose that there exists a path from node u to node v in $MST(G)$ and the direct edge $(u, v) \notin MST(G)$. Let the weight of edge (u, v) be 2^x. This means that the path from nodes u to v consists of edges with weight at most $\leq 2^{x-1}$. By Observation 1, the path from node u to v through the $MST(G)$ would have a cost $\leq 2^x - 1$, which is strictly less cost.

A formal proof: Any shortest path in an undirected graph will traverse an edge at most once. For every non-MST edge $e = (u, v)$, all edges on the MST path between u, v appeared earlier, so their total weight is less than w_{uv}. Therefore, any path containing edge (u, v) can be replaced by the MST path from u to v, and strictly decrease its cost. Repeating this process eliminates all non-MST edges. □

Solution

1. Greedily build the MST of G. If an edge appears in the input that connects disconnected components $u, v \in G$, then unite components u, v. This can be implemented using Kruskal's Algorithm, and save the edge $(u, v, 2^x)$ as an edge of the MST of G.
2. Let $M = MST(G)$. We compute all shortest paths from node 1 by doing DFS on M from node 1 and summing all edges along the path 1 to v (for $2 \leq v \leq n$).

Complexity

$\mathcal{O}(m \cdot \alpha(n) + n)$ time, where m and n are the number of edges and vertices respectively. $\mathcal{O}(m \cdot \alpha(n))$ is the complexity of building the MST, and accumulating path sums takes $\mathcal{O}(n)$ time.

Problem H. Holy Division

Difficulty Estimate	CF 2400
Topics / Techniques	Interactive, Number Theory, Count of Divisors
Solved by	2 participants
First Solve	Swe Hon `swehoneycodes` (88 minutes)

Credits. This problem went through many iterations. The query limit was initially set to 20. Yu Peng first observed that the information from multiple `gcd` queries can be accumulated and removed using just one final `div` query. We later realised that allowing too many `gcd` queries made the problem vulnerable to simple greedy prime-packing heuristics. Through some experiments, we found that Tze Kin's algorithm worked using only 7 queries but we had no proof. We eventually found a concrete proof for 7 queries total, which became the final query limit.

Separately, Swe Hon `swehoneycodes` provided us with a solution in 5 queries; do check out his solution as well!

Let $\tau(x)$ denote the number of divisors for an integer x. The prime factorization of x and the formula for $\tau(x)$ are defined as follows:

$$x = \prod_{i=1}^{k} p_i^{e_i}, \quad \tau(x) = \prod_{i=1}^{k} (e_i + 1)$$

Note that each p_i denote distinct primes in the prime factorisation of x.

Observation 1. *For $\tau(x) \leq 8$, x can have at most 3 distinct prime factors.*

- When x has 3 distinct prime factors i, j, k and $e_i = e_j = e_k = 1$, then $\tau(x) = 2^3 = 8$
- If x has 4 or more prime factors, then $\tau(x) \geq 2^4 = 16$.

Observation 2. *It is sufficient to use at most 1 `div` query.*

Proof. Suppose we only make `gcd` queries before our first `div` query.
Let $d_1, d_2, \cdots, d_m$ be candidate values for `gcd` and y_i be the replies : $y_i = \gcd(d_i, n)$. Then

$$t = \text{lcm}(y_1, y_2, \cdots, y_m)$$

must be a divisor of n. Also note that $t \leq n$. Therefore, we can remove all prime powers discovered by previous `gcd` queries in one `div` query. □

Observation 3. *For any number x, a number x cannot have more than 3 prime divisors larger than $x^{1/4}$.*

Proof. Suppose, for the sake of contradiction, that the prime factorisation of x contains 4 primes p_1, p_2, p_3, p_4 (not necessarily distinct) such that $p_1, p_2, p_3, p_4 > x^{1/4}$. The product $p_1 p_2 p_3 p_4 > (x^{1/4})^4 = x$, which is a contradiction. □

Observation 4. *Following Observations 1 and 2, if we take an integer x and divide away all prime factors $\leq x^{1/4}$, the remaining number x' would have at most 8 divisors.*

Proof. Let x' denote the value of x after dividing away all prime factors $\leq x^{1/4}$.

- If x' has no prime factors larger than $x^{1/4}$ then $\tau(x') = 1$.
- If x' has 3 distinct primes larger than $x^{1/4}$, then $\tau(x') = 8$.
- If x' has 2 distinct primes and one of them is a square of a prime, then $\tau(x') = 2 \times 3 = 6$.

You can work out the remaining cases to see that $\tau(x')$ will only get smaller. □

Because $1 \leq n \leq 10^8$, if we divide away all prime factors $\leq (10^8)^{1/4} = 100$ we would get a number that has at most 8 divisors. Now we need to think about how we can do that.

Observation 5. *A number $n \leq 10^8$ can at most have 8 distinct primes.*

Proof. We maximize the number of distinct prime factors by using the smallest primes, and that they all have exponents = 1.

$$2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 = 9699690 \leq 10^8$$

but

$$2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 = 223092870 > 10^8$$

You can write some code locally to check this. □

Observation 6. *We need 3 `gcd` queries to find all primes ≤ 100 that are in the prime factorisation of the hidden integer n.*

By writing a small routine to get all prime factors ≤ 100, we can build

$$d_1 = 2 \times 3 \times \cdots \times 47 \leq 10^{18},$$

$$d_2 = 53 \times 59 \times \cdots \times 89 \leq 10^{18},$$

and

$$d_3 = 97 \leq 10^{18}$$

This is not the only way to group the numbers.

Observation 7. *We do not need to remove all small primes $p \leq 100$. We only need to remove the full powers of the 5 smallest prime factors from n.*

Proof. Suppose a number n has 5 of its smallest primes removed, then n' after removing 5 smallest primes must satisfy

$$n' \leq \frac{10^8}{2 \times 3 \times 5 \times 7 \times 11} = 43290 \tag{6}$$

Let S denote the number of **non-distinct** prime factors any number $n' \leq 43290$ can have, given that each prime $p \geq 13$.

We claim that $S \leq 4$. The maximum possible number of prime factors is obtained by using as many copies of the smallest allowed prime. So we find that $13^4 = 28561 \leq 43290$ but $13^5 = 371293 > 43290$.

We now show that n' can have at most 8 divisors.

- **Case 1**: n has 8 distinct primes.
  This means n' would have 3 distinct primes. If n' has 3 distinct primes, we will only have 1 copy of each prime. This is because $13 \times 17 \times 19 = 4199$, but $13^2 \times 17 \times 19 = 54587$.

  Therefore, we can conclude that, if n' has 3 distinct primes and contain only prime factors $p \geq 13$, they all must have exponents 1. In this case, $\tau(n') = 8$.

- **Case 2**: n has < 8 distinct primes. This means that n' has at most two prime factors and it is not possible to get more than 8 divisors. In other words, let p_1 and p_2 be two prime factors with exponents e_1 and e_2. Then the following problem has no valid solution for e_1 and e_2:

$$\tau(n') = (e_1 + 1)(e_2 + 1) > 8$$

subject to:

$$\begin{aligned} & e_1 + e_2 \leq 4 \\ & e_1, e_2 \geq 0 \\ & p_1^{e_1} p_2^{e_2} \leq 43290 \end{aligned} \tag{7}$$

Note that $13^2 \times 17^2 = 48841 > 43290$, so its not valid. Therefore, in this case $\tau(n') \leq 8$.

From this observation, we know we only need to remove 5 smallest primes from the factorisation of n. □

Solution

The following solution uses 7 queries.

1. Use 3 `gcd` queries to obtain all distinct primes ≤ 100. By observation 4, we are guaranteed that we have at most 8 distinct primes.
2. Let p_1, p_2, p_3, p_4, p_5 be the 5 smallest primes $p_i \leq 100$ found in step 1, and let x_i be the maximum exponent for prime p_i such that $p_i^{x_i} \leq 10^8$, for $1 \leq i \leq 5$.
3. Use 3 `gcd` queries to find the prime powers of the first 5 primes: We can make these 3 `gcd` queries:
   (a) $y_1 =$ `gcd` $p_1^{x_1} \cdot p_2^{x_2}$
   (b) $y_2 =$ `gcd` $p_3^{x_3} \cdot p_4^{x_4}$
   (c) $y_3 =$ `gcd` $p_5^{x_5}$

   Note that each query is bounded 10^{18}, and we need to simulate for at most 5 primes less than 100.
4. Finally, query `div` $y_1 \cdot y_2 \cdot y_3$. So we use 1 query here.

Alternative Solution

Credit: **Tze Kin**

Observation 8. *If we discovered that a prime p divides n, then repeated squaring allows us to recover high powers of p quickly.*

Suppose the current query contains p^a, and the gcd query returns a number divisible by p^a. After that, we square the returned value and use it in a future `gcd` query.

So the exponent of p that we test grows roughly as

$$1, 2, 4, 8, 16, 32, \cdots$$

Since $n \leq 10^8$, the largest possible exponent is 26 (ie. $2^{26} < 10^8 < 2^{27}$), at most 6 useful squaring rounds are enough to recover the full exponent.

Solution

This is a solution Q `gcd` queries and 1 `div` query. By Observation 8, $Q \geq 6$ (since the maximum exponent of 2 is 2^{26}. We found that $Q = 6$ is enough `gcd` queries, and the algorithm uses 7 queries in total.

1. Let Q be the query budget for `gcd` queries: we let $Q := 6$.
2. Maintain a candidate value $c := 1$.

3. Let t be all prime powers that we find from `gcd` queries.
4. Repeat while $Q > 0$:
   (a) Greedily multiply primes from small to large into c while $c \leq 10^{18}$.
   (b) Query $y =$ `gcd` c. Set $Q \leftarrow Q - 1$.
   (c) Update $t \leftarrow \text{lcm}(t, y)$. Note that t and y may not be co-prime.
   (d) Update c to be the square of found prime factors: $c \leftarrow y^2$.
5. Finally query `div` t.

Discussion

By Observation 7, we know that as long as we remove 5 smallest primes p (where $p \leq 100$) from the prime factorisation of n, the remaining number n' has at most 8 divisors. As we greedily multiply primes from small to large, the 5 smallest primes will be the first 5 primes retained by the scan. Proving this algorithm's correctness is equivalent to proving that we can find and remove at most 5 primes under 100 within 6 squaring rounds of the gcd.

Problem I. Infinite Polygon

Difficulty Estimate	CF 2400
Topics / Techniques	Geometry, Convexity, Ternary Search
Solved by	0 participants
First Solve	N/A

Credits. Thanks to jeroenodb for pointing out a numerical stability issue in the original judge implementation, which led us to improve the implementation and update the jury answers.

Observation 1. *Polygon $\mathcal{P}$ is a convex polygon.*

Proof. Let $m(i)$ denote the gradient of segment i connecting points $P_i = (\frac{1}{i}, \frac{1}{i+1})$ and $P_{i+1} = (\frac{1}{i+1}, \frac{1}{i+2})$.

$$m(i) = \frac{\frac{1}{i+1} - \frac{1}{i+2}}{\frac{1}{i} - \frac{1}{i+1}} = 1 - \frac{2}{i+2} \tag{8}$$

As i increases, the gradient magnitude monotonically increases and approaches 1. A segment later in the counter-clockwise order will have a strictly larger gradient than any earlier segment in the counter-clockwise order.

This ensures that $\mathcal{P}$ lies entirely on one side of a half-plane defined by each segment. Thus, we know that $\mathcal{P}$ is convex. □

Proof. **Alternatively,** you can complete the diagram provided in the sample illustration by drawing a few more points, and you can show that any line segment connecting two points on the boundary of $\mathcal{P}$ lies entirely in $\mathcal{P}$. This also proves that $\mathcal{P}$ is convex. □

Observation 2. *If Q is outside P, the closest point lies on the boundary of P.*

Since $\mathcal{P}$ is convex, if Q lies outside $\mathcal{P}$, the minimum distance from Q to P is attained at some point on the boundary of P. The boundary consists of three parts:

1. The horizontal segment from $(0, 0)$ to P_0
2. The vertical segment P_0P_1
3. The Infinite upper chain $P_1, P_2, P_3, \cdots$

The first two segments can be checked directly using Point-to-segment distance. In fact, we can immediately solve the following corner cases: Let the query point be $Q(x, y)$

- $y \leq 0$, the nearest segment has to be $(0, 0)$ to P_0.
- $x \geq 1$, $y > 0$, the nearest segment has to be P_0P_1

So we are left with minimizing the distance from Q to the infinite upper chain.

Observation 3. *Let the distance $d(i, Q)$ be the minimum segment distance to the segment P_iP_{i+1}. The function d is unimodal in i.*

Proof. In Figure 2 and following Observation 1, if we view the segments in clockwise order, this upper hull has strictly decreasing gradients. This means that the upper hull bends only in one direction: after the hull moves away from a Point Q, it cannot bend back towards Q again.

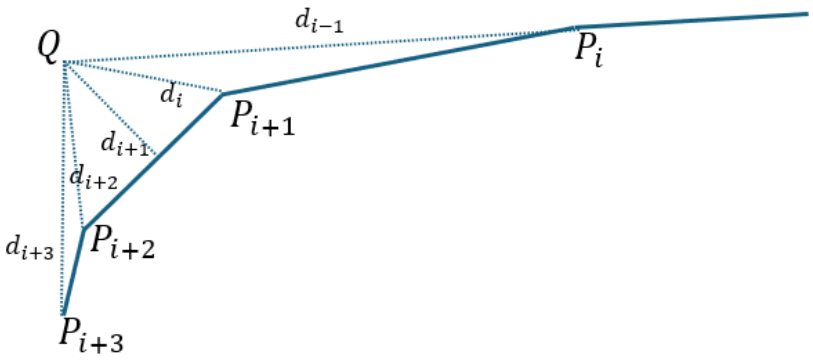

Figure 2: The Points $P_1, P_2, P_3, \cdots$ form the upper hull of the convex polygon $\mathcal{P}$ (See Observation 1)

If Q lies outside P and above the hull, we let the closest point of the upper hull be X. As we move along the upper hull toward X, the distance Q decreases. After passing X, the distance to Q increases (see Figure 2). Therefore,

$$\cdots \geq d(x-2,Q) \geq d(x-1,Q) \geq d(x,Q) \leq d(x+1,Q) \leq d(x+2,Q) \leq \cdots, \tag{9}$$

where x is the segment index that contains Point X.

Therefore, the function is unimodal in segment index i. □

Observation 4. *It is sufficient to consider only finitely many segments of the upper hull.*

Proof. Although the polygon has infinitely many upper-hull segments, the points $P_i = (1/i, 1/(i+1))$ approach the origin O as $i \to \infty$. Therefore, the later segments of the upper hull become arbitrarily close to O, which is already part of the boundary of $\mathcal{P}$.
We can choose a sufficiently large constant S. For every segment P_iP_{i+1} with $i > S$, both endpoints satisfy,

$$0 \leq x \leq \frac{1}{S+1}, \quad 0 \leq y \leq \frac{1}{S+2}$$

Hence every point on every ignored segment is within distance at most

$$\sqrt{(\frac{1}{S+1})^2 + (\frac{1}{S+2})^2} \tag{10}$$

from the origin.

Suppose if there exists a point Q such that the closest endpoint with $\mathcal{P}$ is a segment $i > S$. Replacing i with the origin O increases the distance by $\sqrt{(\frac{1}{S+1})^2 + (\frac{1}{S+2})^2}$.
For an absolute error of 10^{-6}, if we substitute a large value of $S \geq 2 \times 10^6$, we will find the error is less than $\leq 7 \times 10^{-7}$, which is within the tolerance threshold of 10^{-6}.

□

Solution

The solution after derivation works like this: For a given query point Q

1. Check if Q lies inside Polygon $\mathcal{P}$.
   (a) Use the x-coordinate of P to find a segment that covers P. Essentially, solve for the largest segment index i such that
   $$1/i \leq \frac{n_x}{d_x} \Rightarrow i \geq \frac{d_x}{n_x}$$
   (b) Use the `sideOf` check to check which side of the line it is on.
   (c) If Q lies inside $\mathcal{P}$, output `0`.
2. Check if Q has an y-coordinate ≤ 0, then the nearest edge is segment $(0,0) \to p_0$.
3. Check if Q has an x-coordinate ≥ 1, then the nearest edge is the segment p_0p_1.
4. Ternary search over segment indices $i \in [1, S \approx 2 \times 10^6]$ to find the minimizer of $d(i; Q)$. We can assume $|S|$ segments along the boundary.

Complexity

$\mathcal{O}(q \log |S|)$ time, where q is the number of queries, and $|S|$ would be the number of segments you assume to be on the upper hull of $\mathcal{P}$.

Problem J. Judicious Crime Response

Difficulty Estimate	CF 2300
Topics / Techniques	DP (Binary Lifting), Monotonic Stack, Persistent Segment Tree
Solved by	2 participants
First Solve	Pasit `hexahydride` (234 minutes)

Observation 1. *For a station s, if station s chooses to forward the call, the station receiving its calls are fixed.*

Let $NSR(i)$ be $\min\{j > i : r_j < r_i\}$. Using a Nearest Smaller Value algorithm (to the right) we can populate all values $NSR(i)$ in $\mathcal{O}(n)$ time. Note that for this problem, we need $NSR(i)$ to have a strictly smaller r value.

As a result of Observation 1, we get a directed graph of chains, where a station s has a directed edge to $NSR(s)$ and $NSR(s)$ has a directed edge to $NSR(NSR(s))$ and so on. If $NSR(s)$ does not exist, then s has no outgoing edge.

Observation 2. *The distance the car travels is a number-line distance.*

Let the responding station (ie. the station that responds to the call) be x and the crime location be d. We define the coordinate on the number line of location x to be P_x, and we can build it using a prefix sum.

$$P_1 = 0, \quad P_x = \sum_{i=1}^{x-1} t_i = P_{x-1} + t_{x-1} \tag{11}$$

Then the distance from x to d

$$\text{Dist}(x, d) = |P_x - P_d| = \begin{cases} P_d - P_x, & \text{if } x \le d \\ P_x - P_d, & \text{otherwise} \end{cases} \tag{12}$$

Observation 3. *The minimization problem can be made independent of d.*

For each query (s, d), we start off with this objective, where x is the responding station, $\text{chain}(s)$ denotes the set of nodes reachable from s, and $\text{forward}(s, x)$ denotes the total cost of forwarding from node s to node x.

$$\min_{x \in \text{chain}(s)} [\text{forward}(s, x) + r_x + |P_x - P_d|] \tag{13}$$

This is problematic because we must compute solutions for all d. Notice we can split this into two cases

$$\begin{cases} \min_{x \in \text{chain}(s)} [\text{forward}(s, x) + r_x - P_x] + P_d, & x \le d \\ \min_{x \in \text{chain}(s)} [\text{forward}(s, x) + r_x + P_x] - P_d, & x > d \end{cases} \tag{14}$$

Now we get an objective that is independent on d and only dependent on s and x.

Observation 4. *For a node s, we define the x-th "ancestor" from node s as the node reached after traversing x forwarding calls. We can build a recurrence storing the 2^k-th ancestors from node s.*

We use $\text{up}, \text{forward}, g, h$ to store the recurrences:

- Let $\text{up}(s, 2^k)$ denote the 2^k-th ancestor of node s. The recurrence is as follows:

$$\begin{aligned} &\text{up}(s, 2^0) = NSR(s) \text{ if } NSR(s) \text{ exists else inf}, \quad \text{(Base case)} \\ &\text{up}(s, 2^k) = \text{up}(\text{up}(s, 2^{k-1}), 2^{k-1}) \end{aligned} \tag{15}$$

- Let $\text{forward}(s, 2^k)$ denote the total forwarding cost $\text{forward}(s, x)$ where x is the 2^k-th ancestor of s.

$$\begin{aligned} &\text{forward}(s, 2^0) = f_s, \quad \text{(Base case)} \\ &\text{forward}(s, 2^k) = \text{forward}(s, 2^{k-1}) + \text{forward}(\text{up}(s, 2^{k-1}), 2^{k-1}) \end{aligned} \tag{16}$$

- Let $g(s, 2^k)$ denote the minimum value of $[\text{forward}(s, x) + r_x - P_x]$ from node s up to its 2^k-th ancestor.

$$\begin{aligned} &g(s, 2^0) = r_s - P_s, \quad \text{(Base case)} \\ &g(s, 2^k) = \min \begin{cases} g(s, 2^{k-1}), \\ \text{forward}(s, 2^{k-1}) + g(\text{up}(s, 2^{k-1}), 2^{k-1}) \end{cases} \end{aligned} \tag{17}$$

- Let $h(s, 2^k)$ denote the minimum value of $[\text{forward}(s, x) + r_x + P_x]$ from node s up to its 2^k-th ancestor.

$$\begin{aligned} &h(s, 2^0) = r_s + P_s, \quad \text{(Base case)} \\ &h(s, 2^k) = \min \begin{cases} h(s, 2^{k-1}), \\ \text{forward}(s, 2^{k-1}) + h(\text{up}(s, 2^{k-1}), 2^{k-1}) \end{cases} \end{aligned} \tag{18}$$

The total time and space complexity of building the table is $\mathcal{O}(4 \times n \times (\log_2 n))$, where $\log_2(n)$ is because we are defining the recurrence on powers of 2.

Solution

This section discusses the approach using DP and Binary Lifting.

1. Populate $NSR(i)$ for all i using a Nearest Smaller Value algorithm to the right.
2. Build DP tables $\text{up}, \text{forward}, g, h$ discussed in Observation 4.
3. For each query (s, d). Let our answer be denoted as $A = inf$.
   (a) Start at node s, and an accumulated forwarding cost c. Initially, $c = 0$.
   (b) We iterate k from large to small and jump only when $\text{up}(s, 2^k) \leq d$
      i. Before jumping, update
      $$A \leftarrow \min(A, c + g(s, 2^k) + P_d),$$
      where c is the accumulated forwarding cost so far.
      ii. Then update,
      $$c \leftarrow c + \text{forward}(s, 2^k), \quad s \leftarrow \text{up}(s, 2^k)$$
   (c) After the first loop, you are at the last node $s \leq d$ reachable by large jumps. We still need to consider nodes after crossing d, so use
   $$A = \min(A, c + h(s, 2^k) - P_d)$$
   while jumping through remaining ancestors.

Complexity

$\mathcal{O}(n \log n + q \log n)$ time, where all DP tables can be precomputed in $\mathcal{O}(n \log n)$ time and each query is answered in $\mathcal{O}(\log n)$ time.

Fun fact. The crime response theme for this problem was inspired by a conversation Jon had with his girlfriend about the TV show 9-1-1.

Problem K. Killed By Sorting

Difficulty Estimate	CF 1800
Topics / Techniques	Segment Tree / BIT, Constructive Algorithms
Solved by	2 participants
First Solve	Hai Dang `danglayloi1` (233 minutes)

Observation 1. *If we perform cyclic rotations on an array a of length n (assume zero-indexed here), regardless of the number of rotations, every element a_i and its two neighbors positions $i-1 \bmod n$ and $i+1 \bmod n$ remain unchanged under any cyclic rotation.*

Observation 2. *At each iteration, we shift 1 element into its correct position in sorted order. So each operation decreases the size of unsorted suffix of the array. After the i-th iteration, the unsorted suffix only consists of elements larger than i.*

Observation 3. *For $1 \le i \le n-1$, if we sorted the i-th element, for each remaining element in the unsorted suffix, its neighbor becomes the first element larger than i on its left and right in the circular order.*

Proof. Combining Observations 1 and 2, we know that cyclic rotations do not change ordering, but if we remove all neighbors with value $\le i$, the neighbor of some element j is the first elements larger than i on its left and right, for any unsorted suffix of size > 1. □

Consider the sample example $p = [2, 4, 1, 3, 5]$ and we are at `pos` $= 0$. After sorting 1, the resultant array is $p^{(1)} = [1, 3, 5, 2, 4]$, and the unsorted suffix is $[3, 5, 2, 4]$. So the left neighbor of 3 in the unsorted suffix becomes 4, and the right neighbor of 4 becomes 3.

Observation 4. *Let $pos(i)$ denote the position of element i in p (p is not rotated). For $1 \le i \le n-1$, if we sorted the i-th element, the unsorted suffix is the subarray by concatenating all elements in the order that they appear starting at $pos(i)+1$, only if they are larger than i*

Proof. From Observation 3, we know that for every element, the neighbor becomes the first element larger than i on its left and right in the circular order.

If we cyclically shift the array to the left, element i arrives at $\text{pos}(i)$, and the unsorted suffix begins at $\text{pos}(i)+1$. However, if $\text{pos}(i)+1$ was an element less than i, then the unsorted suffix begins at the first element larger than i with position $j \ge \text{pos}(i)+1$ □

Consider the sample example $p = [2, 4, 1, 3, 5]$ and we are at `pos` $= 1$. $\text{pos}(1) = 3$. After sorting 1, we start at $\text{pos}(1)+1 = 4$ and build the unsorted suffix forming $[3, 5, 2, 4]$.

Observation 5. *For $1 \le i \le n$, let s be the starting position of the current unsorted suffix. The number of cyclic shifts at the i-th iteration is equal to the number of remaining elements in the circular interval $[s, pos(i)]$.*

Naively, this is $\mathcal{O}(n)$ to simulate, but we can optimise it in $\mathcal{O}(\log n)$ time if we use a segment tree / BIT to store the count of elements larger than i, discussed in more detail in the solution.

Solution

1. Use a Segment Tree / Fenwick Tree to store the count of value larger than a value i. Initially, all values are present, so all leaf nodes are 1.
2. Create an array $\text{pos}(i)$, that stores the position of element i, $0 \leq \text{pos}(i) \leq n-1$ (zero-indexed).
3. Initialize the starting position of the unsorted suffix, we call it s. Initialize $s = 0$ (zero-indexed)
4. For each iteration $1 \leq i \leq n$
   (a) In the segment tree, update$(\text{pos}(i), 0)$
   (b) Count the number of elements larger than i in the possibly wrapped interval $[s, \text{pos}(i)]$.
      - If $s > \text{pos}(i)$, then the answer is $\text{query}(s, n-1) + \text{query}(0, \text{pos}(i))$
      - If $s \leq \text{pos}(i)$, then the answer is $\text{query}(s, \text{pos}(i))$.
   (c) Set $s \leftarrow (\text{pos}(i) + 1) \bmod n$

Correctness

At each step, we are removing the occurrence of element i by setting the Segment tree value at that position to zero, so the remaining 1s in our range query data structure will always be the ones larger than i.

Complexity

$\mathcal{O}(n \log n)$ time, where n is the length of permutation p.