The 2025 ICPC Caribbean Finals Qualifier

Editorial

Problem set developers

Carlos Joa Fong – SODOCOMP
Marcelo Fornet Fornés – NEAR AI
Rubén Alcolea Núñez – Leil Storage
Tomas Orlando Junco Vázquez – T-Systems Iberia
Anier Velasco Sotomayor – Harbour Space University
José Andrés González Barrameda – Universidad de La Habana
Jorge Alejandro Pichardo Cabrera – Universidad de La Habana
Humberto Díaz Suárez – Universidad de Puerto Rico, Mayagüez

Matcom Online Grader

Leandro Castillo Valdés – ICPC Caribbean
Frank Rodríguez Siret – Harbour Space University
Karel Antonio Gonzalez Zaldivar – Universidad de la Habana

October 9th, 2025

Problem A. Altered States

Author: Jorge Alejandro Pichardo Cabrera – UH
Category: Graph Theory 4, Dynamic Programming 3, Strings 4
Solved by 10 teams

Solution 1

In order to solve this problem, let's first solve a simpler one. Given a single string s of length n, count the number of distinct substrings.

A simpler problem: a single string

The idea is to think of substrings as prefixes of suffixes. That is, the substring $s_{L\ldots R}$ is the $(R-L+1)$-st prefix of the string $s_{L\ldots n}$. Therefore, counting the distinct substrings is the same as counting all different prefixes across all suffixes. Thus, we could do the following:

- Order the suffixes lexicographically. Let $\text{sa}[k]$ be the index of the k-th suffix in this order, for each k.
- Go through all pairs $(\text{sa}[k-1], \text{sa}[k])$ of adjacent suffixes in this order, and add to the count the number of characters in $s_{\text{sa}[k]\ldots n}$ after the longest common prefix between those two suffixes. That's because every common prefix was already counted in the previous iteration, and every new character after the longest common prefix generates a new prefix previously unseen.

Thus, if we were able to sort all suffixes lexicographically and compute the longest common prefix between the pairs of adjacent ones in a reliable time, we could then compute the answer in linear time.

Fortunately, there's a data structure named Suffix Array, which computes these two things precisely; and it can be built in $\mathcal{O}(n \log n)$ time.

Back to the original problem

Back to the original problem, we can imagine the following idea:

- Concatenate all the strings, with a custom separator (for instance, the \$ character).
- Do the previous solution over this new string.

The issue here is that this approach counts substrings that are formed with parts of different strings from the list. To fix it, we just need to be careful to never extend over the first \$ character found in a prefix of any of the suffixes.

Details left as an exercise.

Solution 2

Build a Suffix Automaton on $P = w_1 + \$ + w_2 + \$ + \cdots + \$ + w_n$. Every string of $S(w_1) \cup S(w_2) \cup \cdots \cup S(w_n)$ is in $S(P)$ also every string of $S(P)$ that doesn't contains the character \$ is in $S(w_1) \cup S(w_2) \cup \cdots \cup S(w_n)$, so we can conclude that $|S(w_1) \cup S(w_2) \cup \cdots \cup S(w_n)|$ equals the number of distinct substrings of P that don't contain the character \$.

For any Suffix Automaton, there is a bijection between every path starting from the initial state and every substring from the string it represents(in this case P). What we need to count is the number of substrings not containing \$, this is equivalent to counting the number of paths starting from the initial state that doesn't use \$ transitions. We can count this with a single DFS run, taking into account that the Suffix Automaton is a DAG.

We start in the initial state and keep $Dp[s]$= number of paths starting from s that don't use \$ transitions. Then $Dp[s] = \sum_{u \in N(s)} Dp[u]$ where $N(s)$ is the set of states reachable from s.

The answer is $Dp[initial\_state] - 1$ because you need to exclude the empty substring(path starting and ending in the initial state).

Complexity $\mathcal{O}(\sum_1^n |w_i|)$.

Problem B. Breaking Bad

Author: Jorge Alejandro Pichardo Cabrera – UH
Category: Graph Theory 4
Solved by 1 team

Let's say there are m decorative cells $D_1, D_2, \ldots, D_m$, where each $D_k = (r_k, c_k)$ corresponds to the cell in the row r_k and the column c_k. Let's say they're sorted lexicographically. For a path to be possible, two conditions must hold:

- Any valid path is non-decreasing on the rows and non-decreasing on the columns. For any $j < m$, we must have $r_j \leq r_{j+1}$ and $c_j \leq c_{j+1}$.
- All decorative cells (type 2 and type 4) must lie on distinct diagonals. For any two $i, j : i \neq j$ we must have $r_i + c_i \neq r_j + c_j$.

If either condition is violated, no beautiful path exists, and the sabotage cost is 0.

The problem can be modeled through the min-cut problem. In the min-cut problem, there's a flow network with a source and a sink, and the capacities on each edge corresponds to the cost of eliminating it. A cut is a partition of the set of vertices into two parts, such that the source and the sink are in different parts. The cost of a cut is the sum of the costs of the edges eliminated (those that go from the side of the source to the side of the sink). We can use the classical min-cut problem to model our problem as follows:

- The nodes in the network would correspond to the cells.
- In the problem, what we need to cut are nodes, and the costs are also associated to the nodes. However, the min-cut problem is stated so that the cuts are on the edges. The standard way to handle this to create two artificial nodes $v^{in}_{r,c}$ and $v^{out}_{r,c}$ for each original node $v_{r,c}$. We add edges $v^{in}_{r,c} \to v^{out}_{r,c}$ with capacity $w(r, c)$, representing the cost of fracturing the cell (r, c):

$$w(r,c) = \begin{cases} c_{r,c} & \text{if the cell type is 1 or 2} \\ \infty & \text{if the cell type is 3 or 4} \\ 0 & \text{if the cell type is 0} \end{cases}$$

- We add edges $v^{out}_{r,c} \to v^{in}_{r+1,c}$ and $v^{out}_{r,c} \to v^{in}_{r,c+1}$ with capacity ∞, that represent movement.
- Also, in order to guarantee passing through cells of a diagonal that are decorative, we go through every decorative cell D_k, and forbid passage through any other cell on its diagonal. We achieve that by setting $w(r, c) = 0$ for all cells (r, c) such that $r + c = r_k + c_k$ and $(r, c) \neq D_k$.
- We create a source S and add an edge $S \to v^{in}_{1,1}$ with capacity ∞, given that the path starts at $(1, 1)$, with capacity ∞.
- The path effectively ends after passing D_m. Thus, we create a sink T, and add an edge $v^{out}_{r_m,c_m} \to T$, with capacity ∞.

Now we need to find the min-cut. This is where we apply the max-flow min-cut theorem: the cost of the min-cut is equal to the cost of the max-flow. Hence, we run a max-flow algorithm. The value of the flow found corresponds to the cost of the min-cut. In order to construct the cut, we perform a traversal from the source in the residual graph, going only through edges of positive residual capacity. Any edge (u, v) for which u is reachable and v isn't, belongs to the min-cut.

This graph has $O(n^2)$ nodes, and $O(n^2)$ edges. Thus, an algorithm like Dinic's with capacity scaling is guaranteed to pass. However, this graph is special and has a very specific structure, so that Edmond-Karp's passes too.

Problem C. Circle vs Square

Author: Marcelo Fornet Fornés – NEAR AI
Category: Probability 3, Geometry 2
Solved by 14 teams

There are many solutions for this problem. Let's look at a few things that are different in the circle and the square, and for each of them we can design a specific test:

First solution

In a square of diagonal d, the pair of points furthest away are at distance d, in a diagonal. And if we rotate it $-\pi/4$ or $\pi/4$, we get a direction parallel to one of the sides, in which the longest segment has a length of $\frac{d}{\sqrt{2}}$.

In the case of a circle of diameter d, no matter what angle the diameter is rotated, it always gets to a segment of length d.

Therefore, we can do the following:

1. Find the pair of points furthest away. This can be done in $\mathcal{O}(n \log n)$ time, but a simpler strategy could be to sample 1000 random points from the input, and then we can just compute this in $\mathcal{O}(n^2)$ time.
2. Rotate that line $0°, 45°, 90°, 135°$.
3. On each of those directions, project all points over it, and compute the two points furthest away.
4. Let the minimum diameter across all those 4 projections be d, and the maximum be D.
5. As a statistic, compute $\frac{d}{D}$, which should be ≈ 1 in the case of a circle, and $\approx \frac{1}{\sqrt{2}} \approx 0.7$ in the case of a square.
6. Choose a threshold around 0.85 to discriminate.

Second solution

Let's try to inscribe a circle inside each figure: A square of diagonal d has area $\frac{d^2}{2}$. If we inscribe a circle, this circle has radius $\frac{d}{2\sqrt{2}}$, and hence it has area $\pi\frac{d^2}{8}$. The ratio between these two areas is $\frac{\pi}{4} \approx 0.785$

In the case of circle, the ratio of the areas would be $\frac{1}{2} = 0.5$

Therefore, we can do the following:

1. Find the pair of points furthest away. Let that distance be d.
2. Choose some threshold around 0.7.
3. Count the number of points at distance less than $\frac{d}{2\sqrt{2}}$.
4. If these points represent a percentage of all points, greater than the chosen threshold, then we can infer a square. Otherwise, a circle.

Third solution

There are many ideas similar to the previous one, about inscribing/circumscribing a square or a circle into the figure, and then analyzing things like ratios of areas, diameters, etc. For brevity, we omit them here, but they're all in the same spirit.

Problem D. Doctoral Thesis

Author: Tomás Orlando Junco Vázquez – T-Systems Iberia
Category: Data Structures 2, Number Theory 2, Combinatorics 2
Solved by 51 teams

The following observations summarize the key ideas behind the solution:

1. Two words are anagrams of each other if and only if each character appears the same amount of times in both of them. Thus, checking if w is an anagram of $s[l..r]$ reduces to comparing their character counts.

2. The number of distinct anagrams of a string can be seen as a problem of **permutations with repetitions**. Let $n_a, n_b, \ldots, n_z$ denote the occurrences of each character in a word, and let $n = n_a + n_b + \cdots + n_z$ be its length. Then, the number of distinct permutations is given by the **multinomial coefficient**:

$$\binom{n}{n_a, n_b, \ldots, n_z} = \frac{n!}{n_a!\, n_b!\, \ldots\, n_z!} \tag{1}$$

3. In both types of queries, **we need to quickly know the counts of each character in any substring of** s, motivating the use of prefix sum arrays indexed by characters.

So, we build a bidimensional prefix sum array RSQ, where RSQ$[p][c]$ stores the number of times the character `c` appears in the prefix $s[1..p]$. Here, $|\Lambda|$ denotes the size of the alphabet (e.g., $|\Lambda| = 26$ for lowercase English letters). This $n \times |\Lambda|$ array maps each prefix and character to its cumulative count. This allows computing the number of occurrences of each character in $s[l..r]$ in constant time:

$$\text{count}(c, s[l..r]) = \text{RSQ}[r][c] - \text{RSQ}[l-1][c]$$

With the prefix sums computed, we can now handle the **INSIDE** queries by simply checking that each character in the alphabet appears the same number of times in the word w given in the query and in the substring $s[l..r]$.

For **COUNT** queries, we must compute the number of distinct anagrams of a substring modulo $M = 998244353$. This uses the multinomial coefficient in (1), along with factorials and their modular inverses. Since M is prime and $n << M$, all factorial inverses exist.

The inverses can be computed through Fermat's little theorem:

$$a^{-1} \equiv a^{M-2} \pmod{M}$$

This takes $O(\log M)$ per value, so naively computing all factorial inverses takes $O(N \log M)$ time, where N is the maximum length of any string s (in this problem, $N = 10^6$). It can be done faster though:

$$(k+1)! \equiv (k+1) \cdot k! \pmod{M}$$
$$(k!)^{-1} \equiv (k+1) \cdot ((k+1)!)^{-1} \pmod{M}$$

Hence, after computing $(N!)^{-1}$ in $O(\log M)$, the remaining inverses follow in $O(N)$.

Complexity Summary:

- Factorials and inverses precomputation: $O(N)$.
- Prefix sums construction: $O(n \cdot |\Lambda|)$.
- Query processing: $O(|\Lambda|)$, plus $O(|w|)$ for INSIDE queries.

Overall complexity:

$$O(N \cdot |\Lambda| + Q \cdot |\Lambda| + \sum_{\text{INSIDE}} |w|).$$

where Q denotes the total number of queries.

Problem E. Exact Steps

Author: Rubén Alcolea Núñez – Leil Storage
Category: Sorting-Searching 2
Solved by 115 teams

This problem is intended to be among the easiest ones of the contest. The observations below can be useful for the solution:

1. Binary search naturally forms a **binary search tree** of midpoints.
   - The root is the middle of $[0, n-1]$, found in 1 step.
   - The children are the middles of $[0, mid-1]$ and $[mid+1, n-1]$, found in 2 steps, and so on.
2. The step s corresponds exactly to the **depth** of a node in this recursion tree.
3. Therefore, the problem reduces to generate all midpoints at depth s in the recursive subdivision of $[0, n-1]$.

Solution 1: Simulate Binary Search Recursively (direct construction)
Write a recursive function to simulate the binary search. This function can have the following signature:

```
@param start  Start of the interval
@param end    End of the interval
@param s      Number of steps required to find the number
void binarySearch(int start, int end, int s);
```

This solution simulates how binary search operates, so it is both intuitive and efficient.

Time Complexity: Each node is visited once. The recursion visits $O(n)$ nodes in the worst case, but since the search tree has height $O(\log_2 n)$, at most $O(2^s)$ nodes are explored. For $s \leq \log_2(n)$, this is bounded by $O(n)$. With $n \leq 100000$, this is efficient.

Solution 2: Iterative BFS on Segments

Similar to **Solution 1**, but using a queue starting from $(0, n-1, 1)$.

The solution works like below:

- For each segment, compute the midpoint.
- Push child segments until depth s.
- When depth s is reached, output that midpoint.

This avoids recursion and can be useful in languages where recursion depth is limited. Time complexity is the same as **Solution 1**.

Solution 3: Direct simulation of binary search steps

This solution directly uses the definition of binary search to determine how many steps are needed to find each position.

- Define a function `find_steps(n, i)` that simulates binary search on indices $[0, n-1]$ and returns the number of steps needed to find position i.
- For each index i in $[0, n-1]$, run this simulation and collect those where the number of steps equals s.

This method directly reproduces the binary search process for each possible position. Whenever a position takes exactly s steps to be found, it corresponds to a node at depth s in the binary search tree. This solution performs up to $O(n \log n)$ operations in each case, which is perfectly fine for the constraints of the problem.

Problem F. Final Shutdown

Authors: Rubén Alcolea Núñez – Leil Storage
Jorge Alejandro Pichardo Cabrera – UH
Category: Dynamic Programming 3
Solved by 2 teams

By simulating this process, one realizes that the set of visited generators forms a contiguous interval, and given that one can teleport to any visited generator in 0 units of time, that means that no matter where we are inside that interval $L \dots R$, we can always move to either x_{L-1} in $x_L - x_{L-1}$ units of time, or to x_{R+1} in $x_{R+1} - x_R$ units of time. Notice that the total amount of time to shutdown all generators in the interval $[L, R]$ is:

$$(x_{L+1} - x_L) + (x_{L+2} - x_{L+1}) + (x_{L+3} - x_{L+2}) + \cdots + (x_{R-1} - x_{R-2}) + (x_R - x_{R-1}) = x_R - x_L \leq 1000$$

Key idea: at any point, the set of shut-down generators forms a contiguous interval $[L, R]$. The next generator to shut down must be immediately adjacent to this interval, either at position $L - 1$ or $R + 1$. Each choice yields a different amount of additional energy cost, depending on the time it takes to move to that generator and the total energy of all generators still active during that period.

Let $f(L, R)$ be the minimum total energy consumed when all generators in the interval $[L, R]$ are shut down. The starting generator s satisfies $f(s, s) = 0$. The transitions are as follows:

$$f(L,R) = \begin{cases} 0, & \text{if } L = R, \\ \min\big(f(L+1,R) + \mathrm{F}(e_L, x_R - x_L),\ f(L,R-1) + \mathrm{F}(e_R, x_R - x_L)\big), & \text{otherwise.} \end{cases}$$

Here, $\mathrm{F}(e_X, T)$ denotes the total energy consumed by a generator with base energy e_X after T time units, considering that energy levels cycle periodically up to a maximum value of MaxEnergy = 1000. The formulation of $\mathrm{F}(e_X, T)$ is left as exercise to the reader :).

Since there are $\mathcal{O}(n^2)$ states and each state requires $\mathcal{O}(1)$ transitions, the total time complexity is $\mathcal{O}(n^2)$.

Problem G. Gotcha Station

Author: Humberto Díaz Suárez – UPRM
Category: Graph Theory 3
Solved by 2 teams

The subway network forms an undirected graph where the stations are nodes and the two-way train connections are edges. The problem asks which nodes are required to be on every path between nodes s and e. Equivalently, a node v is a *required node* if and only if removing it from the graph causes e to be unreachable from s. The statement guarantees that the graph is connected, so there will always be a path from s to e. This is a classic graph theory problem with several solutions.

Naive Solution

We loop over the node indices from 1 to n. For each node v, we attempt to find a path from s to e while forbidding passing through v. Performing DFS to find this path takes $\mathcal{O}(n+m)$ time, for n nodes and m edges. Since we do this for $\mathcal{O}(n)$ nodes, the total time complexity is $\mathcal{O}(n(n+m))$.

While straightforward, this approach is too inefficient for 10^5 nodes and $2 \cdot 10^5$ edges.

Fast Solution

An *articulation point* or *cut vertex* is a node whose removal increases the number of connected components in the graph. All of the required nodes on the path from s to e must be articulation points (excluding s and e themselves). However, not all articulation points are required nodes. We can use the Hopcroft-Tarjan algorithm for articulation points (with some extra steps) to find the required nodes efficiently:

```
tarjan_dfs(s)
if parent[e] = UNASSIGNED:
    return UNREACHABLE
required_nodes := list()
required_nodes.append(e)
current := e
while current != s:
    p := parent[current]
    if low[current] >= depth[p]:
        required_nodes.append(p)
    current := p
return required_nodes
```

We apply the Hopcroft-Tarjan algorithm to the input graph, starting the DFS from node s. Let `depth[v]` be the depth of node v in the traversal, let `parent[v]` be the parent of v in the traversal, and let `low[v]` be the lowpoint of v. The algorithm computes these values in $\mathcal{O}(n+m)$ time.

We retrace the path from e to s through parent pointers. For each node v along the path (except s), we get its parent p and compare `low[v]` against `depth[p]`. If `low[v] < depth[p]`, then there must exist a cycle passing through v and p. Removing p wouldn't partition the graph. Otherwise, if `low[v] >= depth[p]`...

- All paths between s and v pass through p.
- Since we are retracing the path from e, we know e is one of v's descendants.
- Therefore, removing p would separate e from s, meaning p is a required node.

This criteria will always classify s as a required node. However, e isn't checked, so it must be added to the results separately. Retracing the path takes $\mathcal{O}(n)$ time. We also have to output the required nodes in ascending order. If we account for sorting, this gives an $\mathcal{O}(n \log n + m)$-time solution.

Note that if the Hopcroft-Tarjan algorithm is implemented recursively, then there is a high chance that it might overflow the call stack on large graphs. A careful implementation in C++ might avoid this, but it's almost guaranteed to occur with Python and Java. The solution is to write an iterative version of the algorithm that relies on a stack.

Bonus

There are other solutions, such as constructing a block-cut tree of the graph. This can be done in $\mathcal{O}(n+m)$ time. The implementation is longer but is conceptually cleaner, so some contestants may prefer it. In addition, this method can be extended with algorithms for lowest-common ancestor (LCA) queries to efficiently find the amount of required nodes between any two endpoints.

Problem H. Hyper Disjoint Set

Author: Marcelo Fornet Fornés – NEAR AI
Category: Data Structures 4, Graph Theory 2
Solved by 0 teams

Let's imagine keeping a DSU with path compression.

In this data structure, it is enough to keep an array p[ ]. Throughout the process of connecting nodes, we need to support:

- Get p[u] for some node u.
- Update p[u] = v for two nodes u and v.
- When applying the DUP operation, notice that if p[v] = u, then p[v + n] = $u + n$. That means that we need to concatenate p[ ] to itself and add $+n$ to the right half of the new array.
- When applying the DUP-link operation, we have something very similar to the case above. But now, since we also add an edge between u and $u + n$, that means that if p[v] = u, then $v + n$ would also be connected to u, which means we can keep p[$v + n$] = u. Hence, there is no need to add $+n$.

Notice that the DUP and DUP-link operations effectively duplicate the number of nodes in the graph. Hence, we only need to process the first $\lceil \log_2(M) \rceil$ DUP and DUP-link operations, where M is the maximum index of a node in any given query. In our case we have $M \leq 10^{18}$, and this means we only process around 60 queries of those kinds.

Now, in order to support those operations, we need a dynamic data structure on top of the array p[ ], which could be a Persistent Dynamic Segment Tree or a Persistent Treap, with lazy propagation. These data structures support these operations in logarithmic time. The number of times we need to look up a position of p[ ] is logarithmic too, and each time, we need a logarithmic number of operations. Hence, we get a solution in $\mathcal{O}(q \log^2 M)$ time.

An alternative approach to using a Persistent Dynamic Segment Tree or a Persistent Treap is to store an array of up to 60 DSUs, each represented by an associative array that maps vertices to their respective parents.

- For each `DUP` operation, we create a new DSU and set it as the current version. All previous versions are considered "frozen" and are only used for queries.
- All `LINK` operations are performed on the most recent DSU.
- We can essentially ignore all `DUPLINK` operations: since vertices u and $u + N$ are linked in such operations, we can treat them as "equivalent" for the purpose of determining which connected component they belong to. However, when processing a query, we must translate the vertex ID to its "equivalent" vertex ID in the base graph from which these `DUPLINK`-ed copies were created.
- How do we query on `DUP` copies? Recursively find the root of the corresponding vertex in the previous DSUs, adjusting the vertex ID as needed (i.e., if the input vertex ID belongs to a `DUP` copy, add a corresponding offset to the returned parent to obtain its equivalent ID in this copy). Finally, query the current DSU to find the root of the connected component containing the vertex.

```
def find(u, t):  # find root of u at time t
    if t == 0:
        return 0 # at time t = 0, there is a single node (with id 0)
    N = DSU[t].size()   # size of base graph at time t
    u = u % N    # this translates vertex u to its equivalent
                 # DUPLINK-ed vertex
    if u >= N // 2:
        # u belongs to DUPed copy
        r = find(u - N//2, t-1) + N//2
    else:
        r = find(u, t-1)
    return DSU[t].find(r)
```

- Assuming we use path compression and hash map, the complexity of a query is $\mathcal{O}(min(D, log(M)) \cdot log(q))$, where D is the number of `DUP` operations.

Problem I. Inverse Harmony

Author: Marcelo Fornet Fornés – NEAR AI
Category: Greedy 3, Number Theory 3
Solved by 30 teams

We want to maximize

$$H = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_n}}$$

This means we need to minimize

$$S = \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_n}$$

Let's see what happens if we apply an operation to the position j, that is: the term $\frac{1}{a_j}$ changes to $\frac{1}{a_j+1}$, hence S gets updated by

$$S := S - \Delta \text{ with } \Delta = \frac{1}{a_j} - \frac{1}{a_{j+1}} = \frac{1}{a_j(a_j+1)}$$

Thus, if we apply reductions $\Delta_1, \Delta_2, \ldots, \Delta_k$, we want to maximize $\Delta_1 + \Delta_2 + \cdots + \Delta_k$. In order to do that, we can follow a greedy approach: always apply the operation over the element j with smallest value a_j. We can prove this is optimal via a simple exchange argument (omitted here).

All is left is simulating this greedy algorithm efficiently. Since $k \leq 10^8$, we can't just simulate it operation by operation. Doing so would take $\Omega(k)$ time, which is too much.

To make it more efficient, we can first sort the array, so now $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$. Then, we need to do many operations until $a_1 = a_2$; that takes $a_2 - a_1$ operations. After doing them, we have $a_1 = a_2$, and we need to do operations to make $a_1 = a_2 = a_3$; that takes $(a_3 - a_2) \cdot 2$ operations. After that, we have $a_1 = a_2 = a_3$ and we want to make them equal to a_4; that takes $(a_4 - a_3) \cdot 3$ operations. And we continue doing so, ..., going through t in increasing order from 2 till n, using $(a_t - a_{t-1}) \cdot (t-1)$ operations in order to make all the first t elements equal. We stop as soon as we run out of operations, or reach the end of the array. If we reach the end of the array and still have operations left, that means now all elements in the array are equal, and then we give $\lfloor \frac{K}{n} \rfloor$ to each of the n elements of the array, and then $+1$ to the first $K\%n$ elements; where K is the remaining number of operations left.

Once we know the final value of each a_j, it's all a matter of computing H, which can be done by applying the definition of Harmonic mean with modular arithmetics.

There are some details omitted.

This solution requires sorting the array and then doing a linear scan, computing some formulas to get the number of operations to perform. Thus, the time complexity is $O(n \log n)$.

Problem J. Judicious XORcery

Author: Jorge Alejandro Pichardo Cabrera – UH
Category: Data Structures 3
Solved by 0 teams

For this problem we have an array A, this will be the array of magic values in real time. We can define the array B as the array that would result if, from the array A we move the separator(mixing rod) to the position 1. Also, we define array P as the Prefix Xor of B, i.e., $P[i] = \bigoplus_{j=1}^{i} B[j]$.(This is also the resulting array after moving the separator from 1 to n)

Note that moving the separator to the left or right without making any type 2 operation won't change the array B. Also, note that given an array A with the separator originally in position k, if after several type 1 operations it returns to position k, it will yield the same array A. This can be proved by induction starting from the fact that after moving the separator to the right and then left(or viceversa) the array remains unchanged. From this, we can conclude that:

$$A[i] = \begin{cases} P[i] & \text{if } i \leq k \text{ (k is the actual position of the separator)} \\ B[i] & \text{if } i > k \end{cases}$$

What we are going to do is to maintain the real state of B all the time so we can quickly calculate A. Then we need to manage the type 2 operations so that the changes made to B reflect the real state from A.

So given a type 2 operation and the separator at position k, we have three cases:

- If $p > k$: then just update $B[p] := x$ because this reflects in A with the change $A[p] := x$ and that's what we want.
- If $p < k$: for this case we need $P[p]$ to take the value of x and all the other positions remain the same. We can do the following $B[p] := B[p] \oplus P[p] \oplus x$ this will result in $P[p]$ taking the value of x, but it will affect the subsequent elements of P. For fixing this we need to do $B[p+1] := B[p+1] \oplus P[p] \oplus x$. Note that the effect these two operations will make in A is just changing the value of $A[p]$ by x.
- If $p = k$ this case is the same as the second except for the "fixing" operation ($B[p+1] := B[p+1] \oplus P[p] \oplus x$), if the separator is in p you can't change the value of $B[p+1]$ because this will imply changing the value of $A[p+1]$, but that's not required. There is no need to fix subsequent elements because $P[p]$ is the rightmost element of P present in A.

You can efficiently maintain P while making updates in B with a Fenwick Tree or Segment Tree.

Complexity $\mathcal{O}((n+q)\log n)$.

Problem K. Kekkaishi

Author: Humberto Díaz Suárez – UPRM
Category: Data Structures 2, Geometry 1
Solved by 61 teams

In this problem, we're asked to find the surface area of the smallest axis-aligned rectangular cuboid such that all points are inside it. Given that the sides are axis-aligned, all we need is the lowest and highest coordinate components among all points in the three axes. Let them be min_x, max_x, min_y, max_y, min_z, max_z respectively. Then, the surface area is:

$$span_x = max_x - min_x$$
$$span_y = max_y - min_y$$
$$span_z = max_z - min_z$$
$$area = 2 \cdot span_x \cdot span_y + 2 \cdot span_x \cdot span_z + 2 \cdot span_y \cdot span_z$$

It's helpful to observe that we can solve the problem independently on each axis. As long as we track the minimum and maximum value on each axis, then we can use the formula above. So the problem can be reduced to maintaining a dynamic minimum and maximum.

There are many approaches:

- Use a C++ multiset to track the min and max. It supports insertions and deletions in $\mathcal{O}(\log n)$ time. This was the simplest and most common solution by far.
- Use two heaps or priority queues per bound (i.e. the min or max on a particular axis). One tracks the max (or min). The other serves as a deletion queue, storing values that must be deleted eventually but that cannot be removed immediately since heaps only support removing the top element efficiently. This approach supports updates in amortized $\mathcal{O}(\log n)$ time. Several teams used it.
- Use a segment tree for each bound. Store a point's components in the trees, then query the min and max values from the trees. Updates run in $\mathcal{O}(\log n)$ time. A handful of teams did this.

There are also more exotic solutions:

- Use a tree that stores points by their IDs. Each node also contains the bounding box that covers all points in its subtree. Merging bounding boxes from subtrees can be done in $\mathcal{O}(1)$ time, so updates run in $\mathcal{O}(\log n)$-time for log-height trees.
- Store points in an octree and maintain a bounding box at each node. Special care must be taken to prevent duplicate points from degrading performance.
- Preprocess the operations. Track the history of min/max values on each axis using segment trees. The details are left as an exercise for readers. :)

A minor implementation detail is being able to remove points by their IDs. This requires a mapping of IDs to points. Even a preallocated array would work since IDs increase sequentially with each insertion. Also, the surface area calculation will overflow when using 32-bit integers since the largest surface area could be $12 \cdot 10^{16}$.

Assuming $\mathcal{O}(\log n)$-time updates and queries, solutions run in $\mathcal{O}(n \log n)$ time (for n spirit operations).

Problem L. Lost Score

Author: Marcelo Fornet Fornés – NEAR AI
Category: Ad-Hoc 1
Solved by 208 teams

Since there are only three kids and three problems, a possible solution is to brute force all possible pairs of kids that solved each of the three problems and check which of these pairings match the given scores of two of the kids. The answer is the score of the third kid.

But, a more clever approach is to realize that there were a total of 14 earned points (since two kids solve the problem worth 1 point, two kids solve the problem worth 2 points, and two kids solve the problem worth 4 total). Therefore, knowing the scores a and b of two of the kids, the score of the third kid is $14 - a - b$.

Problem M. Matrix Operations

Author: Marcelo Fornet Fornés – NEAR AI
Category: Ad-Hoc 1
Solved by 162 teams

For this problem it's enough to just simulate all the operations as they're given. Reversing a row takes $\mathcal{O}(m)$ time. Reversing a column takes $\mathcal{O}(n)$ time. Thus, the time complexity is $\mathcal{O}(q \cdot (n + m))$