2022 North America Championship Solutions

The Judges

May 29, 2023

Problem A: Allergen Testing

Problem

- You are allergic to exactly one of n compounds. You have k sites where you can apply compounds. You repeat the following process d times:
  1. Apply each compound to some (possibly zero) of the sites.
  2. Observe which sites have an allergic reaction.
- Given that each site can only be observed on one day, and a site can not be used if an allergic reaction is observed there, find the minimum number of sites you'll need to discover which of the n compounds you are allergic to.
- Bounds: $1 \leq n, d \leq 10^{18}$.

Problem A: Allergen Testing

Observation: k must be small!

- We will first show that $k \leq 60$ for all valid test cases.
- The maximum possible value of k is clearly obtained by setting $d = 1$ and $n = 10^{18}$ since giving more days or fewer compounds cannot increase the answer.
- In this case, if we label the sites from 0 to 59 and the compounds from 0 to $10^{18} - 1$ and we apply compound x to exactly the sites y where the y-th bit is turned on in the binary representation of x, then we can uniquely determine which compound we are allergic to by reading off the sites that demonstrate an allergic reaction in binary.

Problem A: Allergen Testing

Searching for the answer

- Because $k \leq 60$, if we can quickly check whether it is possible to differentiate among n compounds for some fixed value of k, we can do either a linear scan or a binary search to compute the minimum possible value of k.
- Therefore, we must now answer the question - given d times to repeat the process and k available sites, how many compounds can we differentiate among?

Problem A: Allergen Testing

How many compounds can we differentiate?

- We will prove that for d rounds of the process and k sites, we can differentiate among $(d+1)^k$ compounds by induction on d.
- The base case is $d = 0$. In this case, it is clearly impossible to differentiate between two different compounds since we cannot test either of them. If there is exactly one compound, we don't need any testing to know that we are allergic to it.
- Assume that for $d' < d$ rounds of testing and k sites, we can differentiate among $(d'+1)^k$ compounds.
- Imagine that we have k sites and d rounds. For a fixed subset of x sites, we can apply at most d^x compounds to that site by the induction hypothesis.
- Therefore, we can differentiate among $\sum_{x=0}^{k} \binom{k}{x} d^x$ compounds.

Problem A: Allergen Testing

Solution

- Note that $\sum_{x=0}^{k} \binom{k}{x} d^x = \sum_{x=0}^{k} \binom{k}{x} d^x 1^{k-x} = (d+1)^k$, as desired.
- This completes the induction proof.
- It remains to compute the minimum possible value of k such that $(d+1)^k \geq n$.
- This can be done either naively with Python or by carefully multiplying with 64-bit integers in C++ and Java to detect overflow.
- Possible edge case: When $n = 1$, the answer is zero.

Problem B: A Tree and Two Edges

Problem

- Given an undirected connected graph of n vertices and $n+1$ edges, answer q queries of the form: given two distinct vertices u and v, count the number of distinct simple paths between them.
- Bounds: $n, q \leq 5 \cdot 10^4$.

Problem B: A Tree and Two Edges

Counting number of paths on a tree + one edge

- By definition, there is exactly one simple path between two distinct vertices on a tree.
- If we add an additional edge (a, b) to the graph, this adds two more potential simple paths:
  - Going from u to a along the original tree, traversing the edge from a to b, then going from b to v.
  - Going from u to b along the original tree, traversing the edge from b to a, then going from a to v.
- In order for these paths to be valid, the two segments before and after the edge traversal must be disjoint.

Problem B: A Tree and Two Edges

Validating that two paths on a tree are disjoint

- We can naively check if two paths are disjoint by enumerating the vertices on both paths and seeing if any vertex is common to both. This is too slow for the given problem.
- Note that any path on a tree involves ascending up the tree and descending down the tree. We can partition any path into two subpaths that only ascend by finding the least common ancestor. This is doable in $\mathcal{O}(\log n)$ time.
- We now need to determine if two paths that both only ascend have any common vertices.
- This can also be done by doing an LCA query, or by precomputing the Euler tour of the tree.

Problem B: A Tree and Two Edges

Solving the original problem

- With a tree and two additional edges, there are now eight additional paths to check - there are two ways to order the edges, and each edge can be traversed in either direction.
- We therefore have to check that all three of these segments are pairwise disjoint.

Problem C: Broken Minimum Spanning Tree

Problem

- You are given a labeled spanning tree in a weighted graph. In a single operation, you can remove the label from one of the edges and add a label to another edge, as long as the labeled edges still form a spanning tree.
- Compute the minimum number of operations needed to make the labeled spanning tree a minimum spanning tree of the graph.
- Bounds: $2 \leq n \leq 2 \cdot 10^3, n - 1 \leq m \leq 3 \cdot 10^3$.

Problem C: Broken Minimum Spanning Tree

Solving Simpler Variant - Unique Weights

- Assume for simplicity that all edge weights are unique. The minimum spanning tree is unique in this case.
- If there are e labeled edges not in the minimum spanning tree, then we need at least e operations.
- We can always do this in e operations!
- In a single operation, we pick an arbitrary unlabeled edge that is in the minimum spanning tree and add it.
- This forms a cycle - delete the heaviest edge in the cycle, which by definition can not be in any minimum spanning tree.

Problem C: Broken Minimum Spanning Tree

Solution

- In the original problem, the edge weights are not unique. Therefore, we should pick a minimum spanning tree that maximizes the number of labeled edges initially in it.
- When trying to find a cycle, we can simply DFS from one vertex of the added edge to the other. We know whether a given edge should or shouldn't be in the spanning tree, so as we return from the DFS, we can remove any edge from the spanning tree that isn't in our candidate minimum spanning tree.

Problem D: Fail Fast

Problem

- You are given n automated tests that each have a cost, c, and a probability of passing, p.
- Each test, i, may have a dependency, d, such that test i cannot be executed before test d.
- The cost of executing a sequence of tests is the sum of the test costs up to and including the first test that does not pass.
- The cost of executing a sequence of tests where all tests pass is 0.
- Find a sequence of tests that minimizes the expected cost of running the tests.
- Bounds: $1 \leq n \leq 10^5$, $1 \leq c \leq 10^6$, $0 < p < 1$, p has at most 6 decimal digits.

Problem D: Fail Fast

How would we solve this if there were no dependencies?

- For a sequence, S, let $C_i = \sum_{j=1}^{i} c_j$ and $P_i = \prod_{j=1}^{i} p_j$, where c_j and p_j are the cost and pass probability of the jth test in sequence S.
- The expected cost of a sequence is

$$E(S) = C_1(1 - p_1) + C_2 \cdot P_1(1 - p_2) + \cdots + C_n P_{n-1}(1 - p_n) + 0.$$

- Let S^* be the sequence that minimizes $E(S^*)$.
- We can prove that there is no pair of test indices, i, j in S^* such that $i < j$ and $\frac{c_i}{1-p_i} > \frac{c_j}{1-p_j}$.
- Therefore, S^* can be found by sorting the tests in increasing order of $\frac{c}{1-p}$.

Problem D: Fail Fast

Observation

- In an optimal solution, the remaining test, T, that minimizes $\frac{c}{1-p}$, will be executed as soon as all of its dependencies have executed.
  - If some other test, V, is executed before test T and after T's dependencies have executed, then we can find a better solution by moving T to run before V.

Problem D: Fail Fast

Idea

- Process tests in order of priority. Initially, the priority order will be given by $\frac{c}{1-p}$.
- If we cannot run a test yet, then "merge" it into its parent, to treat this test and its parent as a unit. The new priority value will be given by considering two cases:
  1. The parent test fails before the merged child is run.
  2. The parent test passes and the merged child is able to run.
- So, if the parent had cost c_1 and pass probability p_1 and the child had cost c_2 and pass probability p_2, then replace the parent's cost with $c_1 + p_1 \cdot c_2$ and the parent's pass probability with $p_1 \cdot p_2$. Set its new priority as $\frac{c_1+p_1 \cdot c_2}{1-p_1 \cdot p_2}$.
- Update any test that depended on the child to now depend on the merged unit.
- Keep track of which tests are executed in order as part of a merged unit.

Problem D: Fail Fast

How to do this efficiently?

- Maintain a priority queue of test units sorted by priority.
- Maintain disjoint sets of tests, where tests are in the same disjoint set if they share a parent dependency.
- Only track the updated parent relationships for the root element in each disjoint set.
- Keep track of a linked list of tests in each merged test unit.
- Keep track of which tests have been executed.

Problem D: Fail Fast

The merge operation

- Remove the minimum priority test unit from the priority queue.
- Update the priority value of its parent.
- Take the union of the minimum priority test and its children in the disjoint sets data structure.
- Set the parent of the root element in the disjoint set now containing the minimum priority test unit to be the id of that test unit's parent.
- Concatenate parent's linked list of tests with the minimum priority test unit's linked list of tests.

This operation is $\mathcal{O}(\log n)$.

Problem D: Fail Fast

Algorithm

1. Initialize the data structures.
2. Process the minimum priority test unit.
   1. Check if its dependencies have executed by checking if the parent of the root element in the same disjoint set has been executed.
   2. Choose this test unit to run next if its dependences have been met and remove this test unit from the priority queue.
   3. Otherwise, merge this test unit into its parent.
3. Print each test unit in the order that they are executed. For each test unit, print the tests in its linked list of merged tests before moving onto the next test unit.

Overall time complexity: $\mathcal{O}(n \log n)$.

Problem E: First Last

Problem

- Alice and Bob are playing a word game. From a collection of n words, Alice chooses the first word. Players alternate choosing words. If the other player chose a word ending in letter α, the current player must choose a word starting with α. No word can be chosen more than once, and the first player to not have a valid move loses. Determine who wins if both players play optimally.
- Bounds: $1 \leq n \leq 10^3$, there will be at most three unique letters that can be at the beginning or end of a word.

Problem E: First Last

Reduction to Graph Theory Problem

- Assign each letter a vertex and, for a word that starts with letter α and ends with letter β, draw a directed edge going from α to β.
- From here, we will reframe this problem in graph theory terms - Alice picks a starting vertex and players alternate selecting an edge from the current vertex traveling to the destination vertex for the edge. A player loses if they end up at a node with no unselected edges going out from it. Determine who wins if both players play optimally.

Problem E: First Last

Reducing the Graph

- Claim 1: If there are $x \geq 2$ self-loops for some vertex v, then the result is the same if we remove two of these self-loops.
- Proof: If the current player is in a losing position at vertex v, using a self-loop cannot change the result since the other player can use another self-loop and the current player is back where they started, just with two fewer self-loops.
- Claim 2: If there is an edge from u to v and another edge from v to u, then the result is the same if we remove both of these edges.
- Proof: If the current player is in a losing position at u and they use the edge from u to v, the other player can use the edge going from v to u to send the current player back to where they were.

Problem E: First Last

Solution

- The reduced graph now only has at most three self-loops and no cycles of length 2, so it only has cycles of length 3.
- With memoization, we can compute for all such graphs and edge counts which positions are winning and which positions are losing.
- In practice, it looks like the complexity of this approach is $\mathcal{O}(n^3)$. However, the actual number of positions is far lower due to the small number of self-loops that interrupt traversing the cycle of length 3.

Problem F: Four Square

Problem

- You are given four rectangles. Can they be combined to form a square without any empty gaps or overlaps?
- Bounds: $w, h \leq 10^3$.

Problem F: Four Square

Solution

- If the sum of the areas is not a perfect square, the answer is `NO`.
- Otherwise, we can brute force all possible tilings of such a square to see if any of them result in a decomposition into the four given rectangles.
- One observation that may help is that, for every such decomposition, it is always possible to draw an axis-aligned line such that no rectangle crosses the line.

Problem G: Frequent Flier

Problem

- A traveler has a_i flights scheduled on month i of n.
- Over all periods of m consecutive months, the traveler must pay for at least k flights within the m-month period. If some period contains fewer than k flights, the traveler must pay for all such flights.
- Compute the minimum number of flights the traveler must pay for.
- Bounds: $m, n \leq 2 \cdot 10^5$, $a_i, k \leq 10^9$.

Problem G: Frequent Flier

Solution (High-Level)

- Consider each month in order.
- While the traveler has not yet paid for k flights within the current m-month window and there are still flights to pay for, find the most recent flight that has not yet been paid for and pay for it.
- By an exchange argument, we can show that it is optimal to always pick the most recent flight.

Problem G: Frequent Flier

Solution (Implementation)

- Loop over each month, keeping track of how many flights were paid for per month and how many flights are unpaid. Maintain a sliding window of size m for the number of flights paid for so far.
- Maintain a sorted stack of unpaid flights where the topmost element in the stack tracks the number of unpaid flights and month.
- Use the sorted stack to count how many flights to pay for to maintain having paid for k flights. Repeatedly pop off elements from the stack until k flights have been paid for or there are no more unpaid flights in the m-month window.
- Other data structures can be used as well to maintain the unpaid flights like a segment tree or a balanced binary search tree.

Problem H: Game Show Eliminations

Problem

- You're given n contestants in a game show.
- The second best performing person gets eliminated each round.
- Compute the expected placements of all contestants.
- Bounds: $2 \leq n \leq 1\,000$

Problem H: Game Show Eliminations

Observations

- This is a dp problem, the difficulty comes with creating an efficient state.
- There is no way for person $i - k$ to ever beat person i.
- State only needs to know who could possibly get second place, rest can be compressed.
- The number of people who could possibly get second place is at most k, so we only need to keep a bitmask of those k people somehow.

Problem H: Game Show Eliminations

DP State formulation

- Need to keep track of a bitmask of players, 1 is not eliminated, 0 is eliminated.
- We can fix highest two numbered people first. Suppose second highest remaining person is j. We only need to know bitmask of states of persons $j-k+1$ to j. Everything else is determined.
- State is $(below, mask, above)$. This corresponds to following mask (lowest number person is on the left).

$$\underbrace{1\ldots1}_{below}\underbrace{mask}_{k-1}\underbrace{1}_{1}\underbrace{0\ldots0}_{above}\underbrace{1}_{1}\underbrace{0\ldots0}_{\ldots}$$

- This is $O(n^2 2^k)$ states, can reduce this to $O(nk2^k)$ by noticing if $above > k$, we can reuse answer from $above = k$.

Problem H: Game Show Eliminations

DP State Transitions

- To transition, we need to know probability that each person is second place.
- We can do this with another dp, and can cache results so we compute it at most once.
- We can bucket people by fixing the floor of the performance of each person. If there are x people in the same bucket, then any ordering of those people are equally likely.
- Overall, this adds $O(k^3 2^k)$ runtime.

Problem I: Power of Divisors

Problem

- Define $\tau(x)$ to be the number of positive integer divisors of x.
- Given an integer n, compute the smallest positive integer x such that $x^{\tau(x)} = n$, or report that no such integer exists.
- Bounds: $1 \leq n \leq 10^{18}$.

Problem I: Power of Divisors

Solution

- In order for $\tau(x)$ to be valid, n must be a perfect $\tau(x)$-th power!
- Brute force all possible values of $\tau(x)$, and check if the implied value of x has exactly $\tau(x)$ divisors.
- To compute x, we can binary search for the value of x, taking care to avoid overflow when doing integer multiplication.
- To compute $\tau(x)$, we can use $\mathcal{O}(\sqrt{x})$ trial division.
- Edge case: $n = 1$.

Problem J: Repetitive String Invention

Problem

- Given a string s, compute the number of ways you can concatenate two non overlapping substrings to make a repetitive string.
- A repetitive string is a string with even length, and the first half equals the second half
- Bounds: $2 \leq |s| \leq 800$.

Problem J: Repetitive String Invention

Observation

- Handle case where two substrings are equal.
- Handle case where two substrings have different lengths. The longer substring must be of the form ABA, the shorter one is B, where A and B can be any non-empty string (possibly the same).

Problem J: Repetitive String Invention

Solution

- Precompute $lcp[i][j]$ to be the longest common substring starting at index i and j.
- This array can be used to compute case where two substrings are equal directly in $O(n^2)$. Make sure that strings don't overlap.
- For other case, first fix what B is in the longer substring, suppose it is the substring from position a to b. For each other position c, we can create an array saying if we can start shorter substring at that position (e.g. if $lcp[a][c] \geq b - a + 1$).
- For a fixed B, we can brute force what A is by fixing the position d that the longer substring starts with. We need to check if $lcp[d][c + 1] \geq a - d$. If so, then we count positions c that don't overlap, which can be done efficiently with prefix sums.
- Overall runtime is $O(n^3)$, but constant factor is very low (bounds are high so optimized $O(n^4)$ fail).

Problem K: Space Alignment

Problem

- You are given a file with n lines of code and you wish to replace every tab with exactly k spaces.
- The given file must have *consistent indentation* - if a line of code is nested inside p pairs of braces, there must be exactly $p \cdot i$ spaces that precede the first non-whitespace character in the line. i must be the same over all lines.
- Compute the smallest possible positive integer value of k, or report that it is impossible.
- Bounds: $2 \leq n \leq 100$. Each line starts with at most 1000 characters.

Problem K: Space Alignment

Observation

- If there exists some valid solution, then the minimal such solution must have $k \leq 1000$.
- Proof: Consider two lines, one with t_1 tabs and s_1 spaces, and another with t_2 tabs and s_2 spaces. We need $t_1 k + s_1 = t_2 k + s_2$, or $k = \frac{s_2 - s_1}{t_1 - t_2}$. Since $s_i \leq 1000$, $k \leq 1000$.

Problem K: Space Alignment

Solution

- Check each value of k from 1 to 10^3 and validate that the file has consistent indenting. Print the first value of k that works, otherwise print -1.
- Possible edge case: It is not necessarily the case that the first opening brace is matched by the last closing brace, so one may have to scan the entire file beforehand to find a line with a nonzero number of spaces or tabs to compute the value of i for a given value of k.
- It is also possible to solve this problem algebraically without needing to brute force all values of k. This solution, though faster asymptotically, involves more casework depending on how many lines there are that constrain the number of spaces a tab can have.

Problem L: Splitting Pairs

Problem

- Alice and Bob are playing a variant of Nim. Given n piles, the current player picks $k \leq \frac{n}{2}$ nonempty piles and removes them. They then pick k piles, each with at least two stones, and split each pile into two nonempty piles. The player who cannot move loses. Determine who wins given optimal play.
- Bounds: $n \leq 50$, each pile contains at most 10^{12} stones.

Problem L: Splitting Pairs

Observations

- The only losing configuration is when all piles have exactly one stone.
- When n is even, analyzing small cases we see that a state seems to be winning if and only if at least one pile has an even number of stones.
- When n is odd, we see that having only piles with an odd number of stones is a sufficient but not necessary constraint for the state to be a losing state. Dividing through values by the largest power of 2 that divides all values, it seems to be the case that the state is losing if and only if all piles have an odd number of stones after this reduction.

Problem L: Splitting Pairs

Proofs

- For the even n case, if all piles have an odd number of stones, it is impossible after one move to still only have piles with an odd number of stones.
- If there are no more than $\frac{n}{2}$ piles with an even number of stones, we split those piles into odd-odd combinations and remove just as many odd piles.
- If there are more than $\frac{n}{2}$ such piles, we split $\frac{n}{2}$ such piles into odd-odd combinations and remove the other ones.
- In the odd n case, we can see that if 2^p divides all piles and 2^{p+1} does not, it is impossible to return to a state where 2^x divides all piles and 2^{x+1} does not.
- Let 2^p be the largest power of 2 that divides all piles. Since some pile is not divisible by 2^{p+1}, there are at most n piles that are divisible by 2^{p+1}. We can use the same logic in the even case to split piles into 2^p and $x - 2^p$ to return to a losing state.

Problem M: Who Watches the Watchmen

Problem

- There are n sentries in 3D space, each facing some direction.
- We want every sentry to be pointing at some other sentry, and for each sentry to be pointed at by exactly one sentry.
- With cost 1, we can change the direction some sentry points in.
- With cost 1 000, we can change the location of some sentry as long as it doesn't overlap with any other sentry. The direction does not change in this operation - we need to incur an additional cost of 1 to change its direction as well.
- Compute the minimum cost to attain the desired state, or report that it is impossible.
- Bounds: $n \leq 500$.

Problem M: Who Watches the Watchmen

Initial Observations

- When $n = 1$, it is obviously impossible.
- Otherwise, it's always possible. We only need to move a sentry in one case, when n is odd and all other sentries are collinear.
- If n is even or some sentries are not collinear, we have a minimum cost matching problem - we wish to minimize the number of sentries to rotate to meet the desired condition. Connect sentry i to j with a zero-cost edge if i can see j right now, otherwise connect with a cost of one if there is no other sentry on the line segment connecting i and j.

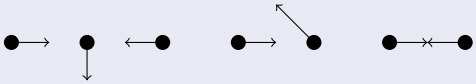
Problem M: Who Watches the Watchmen

The Collinear Case

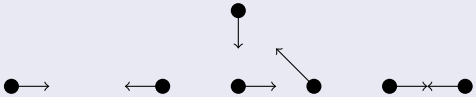
- When n is odd and all sentries are collinear, we must move one sentry. Since moving a sentry is more expensive than rotating all sentries, it is never optimal to move more than one sentry.
- Brute force which sentry to move out of the line. When we move a sentry, we could form a cycle with some subsegment of sentries and the moved sentry, all other sentries must be paired off. We need 3D ray intersection to determine the cost of that cycle. With prefix sums, we can compute for a fixed prefix of sentries the minimum cost needed such that all of those sentries are in some collection of paired cycles.

Problem M: Who Watches the Watchmen

Before Moving



After Moving



Total cost: 1 000.