2025 North America Championship Solutions

The Judges

May 26, 2025

A Totient Quotient

Problem

Given a positive fraction $\frac{a}{b}$ with $1 \leq a, b \leq 10^4$, find a "minimal" pair of integers m, n with $\frac{a}{b} = \frac{\varphi(m^2)}{\varphi(n^2)}$. Minimality is defined by no square of a prime dividing both m and n, and that no prime divides them with the same power.

A Totient Quotient

Problem

Given a positive fraction $\frac{a}{b}$ with $1 \le a, b \le 10^4$, find a "minimal" pair of integers m, n with $\frac{a}{b} = \frac{\varphi(m^2)}{\varphi(n^2)}$. Minimality is defined by no square of a prime dividing both m and n, and that no prime divides them with the same power.

Small cases

- Consider $a = p^2, b = 1$ for any prime p: then, the answer is $m = p^2, n = p$ since $\varphi(p^4) = p^3(p-1)$ and $\varphi(p^2) = p(p-1)$ (can also generalize to $a = p^{2k}, b = 1$).

A Totient Quotient

Problem

Given a positive fraction $\frac{a}{b}$ with $1 \leq a, b \leq 10^4$, find a "minimal" pair of integers m, n with $\frac{a}{b} = \frac{\varphi(m^2)}{\varphi(n^2)}$. Minimality is defined by no square of a prime dividing both m and n, and that no prime divides them with the same power.

Small cases

- Consider $a = p^2, b = 1$ for any prime p: then, the answer is $m = p^2, n = p$ since $\varphi(p^4) = p^3(p-1)$ and $\varphi(p^2) = p(p-1)$ (can also generalize to $a = p^{2k}, b = 1$).
- Consider $a = 2, b = 1$: then, the answer is $m = 2, n = 1$ since $\varphi(2^2) = 2$, $\varphi(1^2) = 1$.

A Totient Quotient

Problem

Given a positive fraction $\frac{a}{b}$ with $1 \leq a, b \leq 10^4$, find a "minimal" pair of integers m, n with $\frac{a}{b} = \frac{\varphi(m^2)}{\varphi(n^2)}$. Minimality is defined by no square of a prime dividing both m and n, and that no prime divides them with the same power.

Small cases

- Consider $a = p^2, b = 1$ for any prime p: then, the answer is $m = p^2, n = p$ since $\varphi(p^4) = p^3(p-1)$ and $\varphi(p^2) = p(p-1)$ (can also generalize to $a = p^{2k}, b = 1$).
- Consider $a = 2, b = 1$: then, the answer is $m = 2, n = 1$ since $\varphi(2^2) = 2$, $\varphi(1^2) = 1$.
- If $a = p, b = 1$ for any odd prime p, then the answer has m divisible by p, n not divisible by p, and all other primes dividing m and n less than p.
    - Similar statement for $a = p^{2k+1}, b = 1$.

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.
- Two cases (assuming $\nu_p(a) > \nu_p(b)$, but can do the other cases symmetrically):
  - Suppose $\nu_p(a) = 2k, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by p, and divide a by p^{2k}.

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.
- Two cases (assuming $\nu_p(a) > \nu_p(b)$, but can do the other cases symmetrically):
  - Suppose $\nu_p(a) = 2k, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by p, and divide a by p^{2k}.
  - Else, suppose $\nu_p(a) = 2k + 1, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by 1, divide a by p^{2k+1}, and *multiply* b by $p - 1$.
- **Proof:** Induction on the maximum prime dividing ab, since each iteration decreases the maximum prime dividing ab.

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.
- Two cases (assuming $\nu_p(a) > \nu_p(b)$, but can do the other cases symmetrically):
  - Suppose $\nu_p(a) = 2k, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by p, and divide a by p^{2k}.
  - Else, suppose $\nu_p(a) = 2k + 1, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by 1, divide a by p^{2k+1}, and *multiply* b by $p - 1$.
- **Proof:** Induction on the maximum prime dividing ab, since each iteration decreases the maximum prime dividing ab.
- Need to keep track of prime factorization of $\frac{a}{b}$ to do iterations (since numbers can grow quickly)

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.
- Two cases (assuming $\nu_p(a) > \nu_p(b)$, but can do the other cases symmetrically):
  - Suppose $\nu_p(a) = 2k, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by p, and divide a by p^{2k}.
  - Else, suppose $\nu_p(a) = 2k+1, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by 1, divide a by p^{2k+1}, and *multiply* b by $p - 1$.
- **Proof:** Induction on the maximum prime dividing ab, since each iteration decreases the maximum prime dividing ab.
- Need to keep track of prime factorization of $\frac{a}{b}$ to do iterations (since numbers can grow quickly)
  - Iteration p takes time $O(\log p)$ after precomputing primes (to calculate factorization of $p - 1$)

A Totient Quotient

Solution

- Iterate primes dividing a or b in decreasing order, beginning with $(m, n) = (1, 1)$.
- Two cases (assuming $\nu_p(a) > \nu_p(b)$, but can do the other cases symmetrically):
  - Suppose $\nu_p(a) = 2k, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by p, and divide a by p^{2k}.
  - Else, suppose $\nu_p(a) = 2k + 1, \nu_p(b) = 1$. Then multiply m by p^{k+1}, n by 1, divide a by p^{2k+1}, and *multiply* b by $p - 1$.
- **Proof:** Induction on the maximum prime dividing ab, since each iteration decreases the maximum prime dividing ab.
- Need to keep track of prime factorization of $\frac{a}{b}$ to do iterations (since numbers can grow quickly)
  - Iteration p takes time $O(\log p)$ after precomputing primes (to calculate factorization of $p - 1$)
- Total time at most $O(\max(a, b) \cdot \log \max(a, b))$.

Circle of Leaf

Problem

- You are given a tree of N nodes rooted at node 1.
- An extra edge is added from every leaf to the root.
- Count the number of distinct spanning trees that can be constructed by removing some subset of edges of this new graph.
- Bounds: $N \leq 2 \cdot 10^5$

Circle of Leaf

Solution

- If the number of leaves is L, we need to remove L edges in order to return the resulting graph into a tree. We also need to ensure that the resulting graph has no cycles.
- Define a leaf path from a node to be a continuous path that can be taken from a node to the root via some leaf node (and its corresponding leaf edge). If a node has ≥ 2 distinct leaf paths, then a cycle will be formed using any two of these paths.
- Consequently, every node must have ≤ 1 leaf paths, and edges must be removed in order to break paths up before a node is reached. This motivates us towards a dynamic programming solution where for each node, we keep track of how many ways that it can either have 0 or 1 leaf paths that still exist.
- Let this quantity be $dp[i][j]$ for node i having $0 \leq j \leq 1$ leaf paths.

Circle of Leaf

Solution

- Suppose we know our *dp* quantities for the children *c* of node *i*. Computing $dp[i][0]$ is

$$dp[i][0] = \prod_{j \in c} (dp[j][0] + dp[j][1])$$

- This is because we can either have no leaf paths from node *j* or we can have 1 leaf path and we remove the edge between *i* and *j*.
- Computing $dp[i][1]$ is a little trickier. We can let the one leaf path from any direct child *j* be the leaf path for node *i*. We can represent this as

$$dp[i][1] = \sum_{j \in c} \left(dp[j][1] \cdot \prod_{k \in c, k \neq j} (dp[k][0] + dp[k][1]) \right)$$

- Computing this is expensive, we can use prefix/suffix products to speed this up.

Circle of Leaf

Solution

- To complete our recursive formula, we can set the base case for any leaf to be:

$$dp[leaf] = [1, 1]$$

- We can either keep the leaf edge and have one leaf path from each leaf, or we can immediately remove the leaf edge and have no leaf paths from our leaf to begin with.
- By computing this recurrence from the leaves upward, our final answer ends up being $dp[0][0]$, since any subset of edges represented by $dp[1][1]$ has a leaf path to the root which forms a cycle.
- The time complexity of this solution is $O(N)$.

Caution

- If using modular inverse to compute the dp transitions, there are test cases that catch solutions that attempt to take the modular inverse of 0. There are many solutions that

Entrapment

Problem

You are given the description of a two-player asymmetric turn-based game, and asked to figure out optimal play for both players, determine the winner, and play as the winning player. Refer to the problems statement for the full rules of the game.

Solution

The full state space of the game is small, so the game can be fully solved. The tricky part is in implementing everything correctly, as you will be writing a lot of code.
Helpful hints:

- The total number of possible boards, queries, and positions is small enough that regular minimax is fine, no alpha-beta pruning required (though memoization is still a good idea).
- Using bitboards can make various computations more compact codewise in addition to being more space and compute friendly.

Geometry Rush

Problem

Starting at $(0, 0)$, you can move diagonally $(1, -1)$ or $(1, 1)$. You are given two polygonal curves, one guaranteed to be above you, and the other below you. For both curves, the points of a curve have non-decreasing x values, and does not self-intersect. You lose if you collide with either curve along the way. The goal is to reach $x = w$ without losing. Output the minimum and maximum y values such that you win, or print `impossible` if you can not win.

Geometry Rush

Solution

Since the curves have non-decreasing x values, we can use a two-pointer solution. We will keep track of the minimum and maximum y values as we move along x, denoted y_{min} and y_{max} respectively. As we increment x, we check every line that includes x and find the minimum/maximum y values that do not collide with the curves, denoted y_{low} and y_{high} respectively. Since you can only move $(1, -1)$ or $(1, 1)$, the parity of x and y must be the same, so we tighten the bounds y_{low} and y_{high} accordingly. Finally, y_{min} can be updated as the maximum of $y_{min} - 1$ and y_{low}, and y_{max} can be updated as the minimum of $y_{max} + 1$ and y_{high}. If at any point, $y_{min} > y_{max}$ it is impossible to win. This gives us a $O(n + m + w)$ solution. The problem can also be done via a linear sweep.

Humans vs AI

Problem

- You are given a constant k and two integer arrays, A and B
- For a given subsequence, let $X = \{i | A_i \geq B_i\}$ and $Y = \{i | A_i < B_i\}$
- You want to count the number of non-empty contiguous subsequences such that $\sum_{i \in X} (A_i - B_i) \geq \sum_{j \in Y} k \cdot (B_j - A_j)$ under the constraint that an adversary can optimally move an element from X to Y or vice versa.
- Bounds: $1 \leq k \leq 100$, $1 \leq |A| = |B| \leq 2 \cdot 10^5$

Humans vs AI

Simpler Variant - No Swaps

- Assume that there is no adversary first
- We can rewrite the property we want to find instead as counting the number of non-empty contiguous subsequences such that $\sum_{i \in X} A_i - B_i - \sum_{j \in Y} k \cdot (B_j - A_j) \geq 0$
- Create a new array C such that $C_i = A_i - B_i$ if $A_i - B_i \geq 0$ otherwise $C_i = k \cdot (A_i - B_i)$. Create another array PC which stores the prefix sums of C.
- This problem now reduces to counting the number of pairs (i, j) such that $i < j$ and $PC_i \leq PC_j$. This can be done many ways; one way is to use some order statistic tree while iterating over PC.

Humans vs AI

Solution

- Adding the adversary back in, we now need the following observation: for a given subsequence, if $|X| = 0$, then the adversary will not swap. Otherwise, the adversary swaps an $i \in X$ such that $A_i - B_i = max_{j \in X}(A_j - B_j)$.
- We can now fix the index s being swapped. This happens over contiguous subsequences such $[i, j]$ such that $i \leq s \leq j$ and s is the smallest index such that $A_s - B_s = max_{k=i}^{j} A_k - B_k$.
- The index being swapped subtracts a constant amount from the subarray with respect to s, which we can factor in when taking the difference of the prefix sum.
- Doing this naively is still $O(n^2)$.

Humans vs AI

Solution cont.

- One way to speed this up is to use some data structure that allows us to do queries of the form: count the number of values in the subarray $[i, j]$ that are greater than or equal to some query value q. This is usually done online using a persistent order statistic tree or a wavelet tree.
- Then, for a given s, let i be the largest index such that $A_i - B_i \geq A_s - B_s$ and $i < s$. Let j be the smallest index such that $A_s - B_s < A_j - B_j$ and $j > s$. We can iterate over the smaller of $[i+1, s]$ and $[s, j-1]$, using the query structure to count the number of subarrays that satisfy our properties.
- By choosing the smaller of the two sides to iterate over, we can bound the number of queries by $\mathcal{O}(n \log n)$.
- Another approach is to store the queries and process them offline, using some smart sorting and a point add, range sum data structure, of which there are many.

Mob Grinder

Problem

- A mob grinder is an $N \times M$ grid consisting of conveyor belts that move the mobs on it up, right, down, or left (depending on their orientation).
- The top-right corner of the grid is empty, and the goal is such that if a mob is on the grid, it will eventually be moved to the top-right corner by the conveyor belts.
- You are given N and M, along with constants U, R, D, L.
- Your goal is to output a valid mob grinder of size $N \times M$ such that it contains U conveyor belts going up, R going right, D going down, and L going left (or state that it is impossible.)
- Bounds: $1 \leq N \cdot M \leq 10^5$, $U + R + D + L = N \cdot M - 1$

Solution

- First, note that if there are fewer than $N-1$ U's or fewer than $M-1$ R's, a solution is impossible.
- Now, consider an arbitrary path from the bottom-left corner to the top-right corner made up of R's and U's that divides the grid into two sections. This path contains exactly $N-1$ U's and $M-1$ R's.
- Key insight: all cells in the top-left section can be filled in arbitrarily with R's and D's, and all cells in the bottom-right section can be filled arbitrarily with L's and U's.
- This gives rise to a solution: first subtract out $N-1$ U's and $M-1$ R's.
- Then, select a path from the bottom-left to the top-right corner that divides the grid into two sections of the correct sizes based on the number of U's, R's, L's, and D's remaining.
- Lastly, fill in the two sections as described earlier.
- The time complexity of this solution is $O(N \cdot M)$.

Most Scenic Cycle

Problem

- You are given a graph $G = (V, E)$, which is biconnected – or as the problem says, *robustly connected*
- G is *regionally connected* - there exists some set of simple cycles F (called *regional cycles*) in G with the following properties (though you are not provided F):
  - $|F| = |E| - |V| + 1$
  - Every edge is part of one or more cycles in F
  - The existence of at least one shared edge between two cycles $a \in F$ and $b \in F$ implies that one and only one unbroken path of connection exists between a and b
  - The dual graph of the regional cycles—where each regional cycle is connected to another if they share an edge—is a tree.
- Each edge in G has an integer weight associated with it
- Find the sum of edge weights, of the cycle in G whose sum of edge weights is maximum

Most Scenic Cycle

Bounds

- $2 \leq V \leq 2 \cdot 10^5$
- $2 \leq E \leq 4 \cdot 10^5$

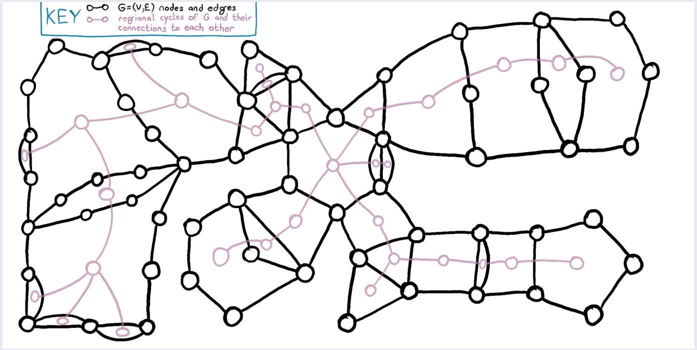
Graph structure notes 1/3

- The fact that there are exactly $|E| - |V| + 1$ cycles in F, and that all edges in E are part of one or more cycles in F, guarantees that F is a *cycle basis*, i.e. a set of cycles which can completely represent all of G's cycles by symmetric differences
- Shorter way of explaining G's connection properties: G is a planar graph whose dual graph (ignoring the "outer region") is a tree.

Most Scenic Cycle

Graph structure notes 2/3

- In the example pictured: black nodes and edges form G; pink nodes and edges are regional cycles of G and their connections with one another



Most Scenic Cycle

Graph structure notes 3/3

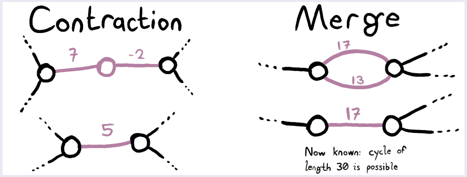
- For purposes of explanation, a biconnected and regionally-connected graph will be called a BCRCG (biconnected, regionally-connected graph).

Most Scenic Cycle

Solution

- Let $G' = G$, and repeatedly simplify G' with two types of operations:
  - Contraction – if a node in G' has two and only two edges of weights a and b to nodes c and d, replace them with a single edge connecting c and d with a weight of $a + b$.
  - Merge – if two edges of weights a and b connect the same pair of nodes in G', record the fact that $a + b$ is a possible cycle length in G, and delete the edge of lesser weight.
- Stop when neither operation can be used on G' anymore. Record self-loop weights as cycle lengths, if any. Output the maximum-recorded cycle length.

Diagram of available operations



Most Scenic Cycle

What the solution does

- Because G''s regional cycles are connected like a tree, some regional cycles are "leaf cycles"
    - Every time a leaf cycle is "merged away", if the only cycle that shared an edge with it becomes a leaf cycle, then *that* cycle can have *its* degree-2 nodes contracted, and *it* can also then be "merged away"
    - Because the dual of G' is a tree, there always exists at least two leaf cycles to "merge away" via our operations, unless one regional cycle remains
    - If one cycle remains, said cycle can either be contracted until it is a self-loop, or, if it has two edges remaining, merged to form a graph of one edge connecting two nodes
- So, the only two ways one stops being able to merge and contract on G', are the only situations in which G' stops being a BCRCG:
    - One node with a single self-edge remains
    - Two nodes remain, with one edge connecting them (*the only exception to an above bullet point on this slide)

Most Scenic Cycle

Time complexity

- Every merge removes one edge -> $O(|E|)$ merges
- Every contraction removes one edge and one node -> $O(|V|)$ contractions
- Every merge between a pair of nodes which leaves only one edge between said pair, makes at most two more contractions possible.
- Every contraction may make a merge possible
- We can use queues and hash tables to efficiently determine what can be merged and what can be contracted
- Expected runtime: $O(|E| + |V|)$

Ornaments on a Tree

Problem

- You are given a tree of N nodes, rooted at node 1.
- Each node either has a value a_i already written on it or is empty.
- You want to assign values to the empty nodes such that for every node, the sum of values on that node and its children does not exceed a constant K, and that the total sum of values in the tree is maximized.
- Bounds: $1 \leq N \leq 2 \cdot 10^5, 0 \leq a_i, K \leq 10^9$

Ornaments on a Tree

Solution

- First, consider the case where there are no pre-written values on the nodes.
- Claim: There exists an optimal solution which has as much value as possible assigned to the leaf nodes.
- To show this, we use an exchange argument. Consider an optimal solution where this is not the case.
- Then, there must be a leaf node with value less than K and whose parent has value greater than 0;
- We can then transfer value from the parent to the leaf to obtain an optimal solution for which this is true.

Ornaments on a Tree

Solution

- Once the leaves have as much value as possible assigned to them, note that their parents now effectively each have a restriction on the maximum value that can be assigned to them.
- We can use a similar exchange argument from earlier to show that there exists an optimal solution with as much value as possible assigned to the parents (given these new restrictions).
- Continuing this argument recursively up the tree naturally gives rise to a solution: greedily assign as much value as possible to the nodes, going from the bottom up.
- Once you reach a parent, you can update all of its children, and from there, their values are fixed.

Ornaments on a Tree

Solution

- How do we deal with nodes with values pre-written on them?
- If the node you're on has a child with a pre-written value, you just treat it as though you can't change it.
- If there's a node with a pre-written value above you, you just take that restriction into account when you process the pre-written node.
- Thus, the algorithm from earlier does not change very much, and the time complexity is still $O(N)$.

Polygon Partition

Problem

- You are given a polygon that doesn't have any lattice points on the boundary.
- Partition semi-integer points on boundary into two sets with equal sum.
- The sum is taken over floor of the non-integer coordinate.

Polygon Partition

Solution

- You can transform this problem from arbitrary polygons to a simplified polygon where:
    - All vertices are at half-integer coordinates (e.g. $(x.5, y.5)$).
    - All edges are axis aligned.
    - The set of n_i of the points on the boundary is the same as the original polygon.
    - Technically, this simplified polygon can intersect itself, but what's important is that it is a closed path.
- This doesn't need to be done explicitly, but it's just used to visualize the solution.

Polygon Partition

Solution

- Consider the polygon as directed counter-clockwise (so outer bottom edge is left to right).
- Consider horizontal and vertical edges separately.
- For horizontal edges, they either go left to right or right to left.
- If we consider any integer vertical line (e.g. a line $x = X$ where X is an integer), it will intersect an even number of horizontal edges.
- These edges will alternatively go right to left and left to right from top to bottom.
- If we consider the taking the n_i values of right to left points, and subtract n_i values of left to right points, we will get a value equal to length of the vertical line inside the polygon.
- Since our polygon is very well-formed, summing this over all integer vertical lines will give us the area of the polygon.

Polygon Partition

Solution

- We have (sum right to left) - (sum left to right) = area of polygon.
- For vertical edges, we can do the same thing.
- We have (sum down to up) - (sum up to down) = area of polygon.
- We can combine these two equations to get:
- (sum right to left) - (sum left to right) = (sum down to up) - (sum up to down)
- This means (sum right to left) + (sum up to down) = (sum left to right) + (sum down to up)
- This is exactly the partition we want, and works for the original polygon.

Popping Balloons

Problem

Given a ternary array S with $n = |S| \leq 2 \cdot 10^5$, randomly remove elements from the array until the result is sorted. Compute the expected number of seconds required until this happens (modulo $998,244,353$).

Popping Balloons

Problem

Given a ternary array S with $n = |S| \leq 2 \cdot 10^5$, randomly remove elements from the array until the result is sorted. Compute the expected number of seconds required until this happens (modulo $998,244,353$).

Some Observations

- When the process ends, it ends on a non-decreasing subsequence of S.

Popping Balloons

Problem

Given a ternary array S with $n = |S| \leq 2 \cdot 10^5$, randomly remove elements from the array until the result is sorted. Compute the expected number of seconds required until this happens (modulo $998,244,353$).

Some Observations

- When the process ends, it ends on a non-decreasing subsequence of S.
- Let X be a random variable dictating the length of the remaining array at termination. Then,

$$\mathbf{Pr}[X \geq k] = \frac{\text{\# of non-decreasing subsequences of length } k}{\binom{n}{k}}.$$

Popping Balloons

Problem

Given a ternary array S with $n = |S| \leq 2 \cdot 10^5$, randomly remove elements from the array until the result is sorted. Compute the expected number of seconds required until this happens (modulo $998,244,353$).

Some Observations

- When the process ends, it ends on a non-decreasing subsequence of S.
- Let X be a random variable dictating the length of the remaining array at termination. Then,

$$\mathbf{Pr}[X \geq k] = \frac{\#\text{ of non-decreasing subsequences of length } k}{\binom{n}{k}}.$$

- Our final answer is $n - \mathbb{E}[X] = n - \sum_{k=1}^{n} \mathbf{Pr}[X \geq k]$.

Popping Balloons

Solution

- Need to compute number of non-decreasing subsequences of each length quickly, but this is not obvious for a general array!
    - Idea: Divide & Conquer FFT, leveraging S being ternary.

Popping Balloons

Solution

- Need to compute number of non-decreasing subsequences of each length quickly, but this is not obvious for a general array!
  - Idea: Divide & Conquer FFT, leveraging S being ternary.
- For a subrange $[a, b)$ and $0 \leq i \leq j \leq 2$, let $f_{a,b}(i, j, k)$ be the number of non-decreasing subsequences of length k contained in $[a, b)$, starting with value i, and ending with value j.
  - Can construct polynomial $f_{a,b,i,j}(x)$ with k'th coefficient being $f_{a,b}(i, j, k)$.
- Then, just need to multiply appropriate polynomials to merge (conquer) two ranges.

Popping Balloons

Solution

- Need to compute number of non-decreasing subsequences of each length quickly, but this is not obvious for a general array!
  - Idea: Divide & Conquer FFT, leveraging S being ternary.
- For a subrange $[a, b)$ and $0 \le i \le j \le 2$, let $f_{a,b}(i, j, k)$ be the number of non-decreasing subsequences of length k contained in $[a, b)$, starting with value i, and ending with value j.
  - Can construct polynomial $f_{a,b,i,j}(x)$ with k'th coefficient being $f_{a,b}(i, j, k)$.
- Then, just need to multiply appropriate polynomials to merge (conquer) two ranges.
- After computing $f_{0,n,\cdot,\cdot}(x)$, add them all together and scale coefficients by $\binom{n}{k}$ to get final answer.

Popping Balloons

Solution

- Need to compute number of non-decreasing subsequences of each length quickly, but this is not obvious for a general array!
  - Idea: Divide & Conquer FFT, leveraging S being ternary.
- For a subrange $[a, b)$ and $0 \leq i \leq j \leq 2$, let $f_{a,b}(i, j, k)$ be the number of non-decreasing subsequences of length k contained in $[a, b)$, starting with value i, and ending with value j.
  - Can construct polynomial $f_{a,b,i,j}(x)$ with k'th coefficient being $f_{a,b}(i, j, k)$.
- Then, just need to multiply appropriate polynomials to merge (conquer) two ranges.
- After computing $f_{0,n,\cdot,\cdot}(x)$, add them all together and scale coefficients by $\binom{n}{k}$ to get final answer.
- Intended time complexity: $O(10n \log^2 n)$ (the constant factor is based on the alphabet size).

SLA Tomography

Problem

- Given a list of integers of length n, representing the amount of resin left at each level of the process (refer to problem statement for more details), output the minimum width of a design that would result in these resin amounts
- Bounds: $1 \leq N \leq 10^5$ and $0 \leq x \leq 10^9$

SLA Tomography

Solution

- We create the design from the bottom up. The design will have resin pools and separating columns that don't have any on top.
- Let us define x as the number of columns that can still have resin added to them without draining away.
- We claim these x columns are in one group. If they are not, we can merge the two groups, remove one separating column, and get a better answer.
- If the resin amount for the next layer is r and $r < x$, then we need to increase the width by 1 as we need to break the pool of x columns into two groups of $x - r$ and r.
- If the resin amount for the next layer is r and $r \geq x$, then we need to increase the width by $r - x$ as we need to add $r - x$ columns to the current group to account for the next layer.

Solar Farm

Problem

Given an integral radius r and integer width and height w and h, find integers i and j such that

- a rectangle of width $i \cdot w$ and height $j \cdot h$ fits inside a circle of radius r, and
- the product $i \cdot j$ is maximized.

Solar Farm

Solution

Define $f(i)$ to be the largest integral value j such that an $i \cdot w$ by $j \cdot h$ rectangle fits in the circle. Then we want to find the i that maximizes $f(i)$.

Concerns:

- While $f(i)$ has a closed form, the bounds are large enough to prevent just calculating this over all possible values of i.
- $f(i)$ is NOT unimodal, so naive ternary search, for example, will not return the correct answer.
- The bounds are also large enough that one must be careful not to run into precision issues; in particular, safe sqrt must be used.

Solar Farm

Solution cont.

Approach:

- Assume WLOG that $w \geq h$.
- Observe that while $f(i)$ is not unimodal, if we define $F(i)$ to be the largest REAL value j whose corresponding rectangle fits in the circle, then $F(i)$ IS unimodal. $F(i)$ is also easy to maximize because if we also relax i to be real, then then the best choice of i is the one that results in a square with diagonal $2r$, and the best integral choice of i has to be either the integer just above this or the one just below this.
- Using the above, we can show that the i that maximizes $f(i)$ and the i that maximizes $F(i)$ are at most $O(\sqrt{r})$ apart.
- Because of this, we can start from the value of i that maximizes $F(i)$ and search on either side within bounds that we calculate either in advance via pen and paper or on the fly via evaluating $f(i)$ and $F(i)$ as part of binary searches.

This Is Sparta!

Problem

- Given an array V, you repeat the following process K times
  1. Sort V in non-decreasing order
  2. Set $V'_1 = V_1$
  3. For every $i \geq 2$ in order, set $V'_i = V_i - V'_{i-1}$

K is far too large (up to 10^{18}) for direct simulation to work, but we can reduce the problem significantly.

This Is Sparta!

Observation 1: Elements reach 0 very quickly

- By brute forcing small cases, you may observe that many elements reach 0 (and can therefore be ignored) after a very small number of iterations. We can show some bounds on this as follows:
    1. Suppose that we had 59 buckets, and we put an element v into bucket i if $2^i \leq v < 2^{i+1}$. Suppose each bucket i has b_i elements in it.
    2. Suppose that elements v_i and v_{i+1} are in the same bucket j and v_i' has already been calculated
    3. Since $v_i' \leq v_i \leq v_{i+1}$ we know that either v_i' is in bucket j or v_i' dropped out of bucket j.
    4. If v_i' is still in bucket j, then v_{i+1}' is guaranteed to drop out of bucket j (imagine the leading bits cancelling each other out). This means that between every pair of elements in the same bucket, at least one will drop.
    5. This means that in a single iteration, $\lfloor \frac{b_i}{2} \rfloor$ elements will drop out of bucket i every iteration.
    6. We can simulate the worst case of this by putting 10^5 elements in bucket 59 and computing the number of iterations before as many elements as possible drop to 0. We see that it take 166 iterations before N reaches 59.

This Is Sparta!

Can we go further?

- K is still too large to effectively simulate, but we can actually find another bound on reducing N even further by constructing worst cases.
    1. Suppose that we had 3 elements and we wanted to reduce down to 2 elements.
    2. If our sequence is $1\ x\ y$, x will always reduce to 0 in at most x iterations.
    3. In order for y to not reduce down to 0 earlier, we need $y - x - (x-1) - (x-2) ... \geq 0$ so we want $x = \sqrt{2y}$.
    4. The maximum value of y is 10^{18} so we can compute this as the worst case
- We can use this small case to inform how we would reduce $N + 1 \to N$, and we can see that an upper bound on the number of iterations to do this will be around $\sqrt[N]{10^{18} N!}$.
- To get from $N = 60$ to $N = 3$, we can set an upper bound of 2000000 iterations.
- Unfortunately, reducing further would take at least 10^9 iterations which is too expensive, even with a small value of N.

This Is Sparta!

Handling N = 3

- Now we need to handle the problem when $N = 3$ and K is large, but we can actually simulate many iteration steps at once.
- Suppose that we have the sequence $[a,\ b,\ c]$ and the order of elements does not change.
  1. After one iteration: $[a,\ b-a,\ c-b+a]$
  2. After two iterations: $[a,\ b-2a,\ c-2b+3a]$
  3. After three iterations: $[a,\ b-3a,\ c-3b+6a]$
- We can observe a very clear pattern: if we have to simulate k iterations where the order of elements doesn't change, we get that our array will always look like $[a,\ b-ka,\ c-kb+\frac{k(k+1)}{2}a]$
- However, this expression can overflow even though the array values cannot. This can be circumvented by using Python, or by rearranging the third term as $c - \frac{k}{2}(2b-(k+1)a)$.
- Finally, since the order of elements can change, we can simulate as many steps as it would take until the order changes, sort the terms again, and repeat this process.

This Is Sparta!

Solution

- Simulate 2000000 "slow" steps, making sure to ignore elements that become 0 during the process.
- At this point, $N = 3$. Simulate "fast" steps until $K = 0$ or $N = 1$. An approximation for the number of iterations before relative order changes is $\min(\frac{b}{a}, \frac{c}{b})$.
- Since this process is similar to the Euclidean Algorithm, this will terminate in $\log \max v$ steps.
- The time complexity looks like $O(166 * N \log N + 2000000 * 60 * \log 60 + \log \max v)$. This might seem slightly higher than what would fit in a 1 second time limit, but not every iteration runs on the same value of N. In fact, using the tighter bounds from earlier slides and writing a program to compute the work for each iteration shows us that the amount of work is less than $2 * 10^8$. Since these bounds are infeasible to achieve in practice, the solution runs comfortably.

This Is Sparta!

Aside

This process will compute the greatest common divisor (gcd) of every numbers since it operates very similarly to the Euclidean Algorithm. This fact was not needed to solve the problem, although it could motivate the solution idea of "fast" steps.