# 2021 North American Qualifier Solution Outlines (ordered by time of first correct solution)

The Judges

Jan 22, 2022

# Eye of Sauron – First solved at 0:01

## Problem

- Given a string containing the substring "()" and a number of vertical bars, determine whether the "()" is centered

## Solution

- Process input one character at a time, counting bars until "()" is encountered
- Begin counting bars again to verify that the number of bars after the parentheses agrees with the count of bars before them
- (In Python, just split on "()" and then compare the lengths of the arrays)

# Mult! – First solved at 0:02

## Problem

- Given: a sequence of positive integers representing one or more rounds of the game "Mult!"
- Each round begins with a target integer $t$ and ends with the next multiple $m$ of $t$ that appears in the sequence
- Print all values of $m$

## Solution

- Set start-flag to "true"
- Loop through input:
  - When start-flag is true, save input to target, reset start-flag to false
  - When start-flag is false and input is a multiple of target, print input, set start-flag back to true

# Common Factors – First solved at 0:07

## Problem

- **Slides for this problem are not yet completed**

# Stamp Combinations – First solved at 0:07

## Problem

Given an array of numbers, determine whether the sum over a subarray from the beginning plus a subarray from the end match a query value $q$.

# Stamp Combinations – First solved at 0:07

## Solution 1

Read in the sequence to an array $A$, and compute a prefix sum. Store these either in a hash map or in an array $B$ (which is ordered and can be binary searched).

For a query value $q$, check each suffix sum, $s <= q$ to see if $q - s$ was a prefix sum.

# Stamp Combinations – First solved at 0:07

## Solution 2

If the sum of all elements in the array is $T$, then if our query is $q$, we want to find whether some subarray sums to $T - q$

We can use the traditional sliding window technique to find whether such a subarray exists.

- Set the initial start and end index of the window before the first element, with the sum at 0
- If the sum of the window is less than the target, increase the end index (and add that new value to the sum)
- If the sum of the window is greater than the target, increase the start index (and subtract that value from the sum)
- Continue until either a sum is matched, or you try to extend beyond the end of the array.

A variation on the sliding window technique can also be used to find the beginning and ending of the array summing to $q$, directly.

# Stamp Combinations – First solved at 0:07

## Warnings and Pitfalls

- Efficient implementation of the problem is important. Inefficient searching, indexing, or extraneous comparisons can yield time limit exceeded.
- Likewise, buffering output to prevent multiple writes may yield faster runtimes.
- Handle the edge case of 0 stamps needed
- Be sure not to pass when there are more stamps requested than are on the roll.

# Tic-Tac State– First solved at 0:09

## Problem

Given a tic-tac-toe state represented as bits, figure out the state of the game. Bits 0-8 are positions that have been played. 9-17 are positions played by X. 18 indicates who plays next.

## Observations

1. There 8 winning patterns: 3 rows, 3 columns, two diagonals. These patterns correspond to these winning bitmasks in binary: `0b111000000`, `0b000111000`, `0b000000111`, `0b100100100`, `0b010010010`, `0b001001001`, `0b100010001`, `0b001010100`
2. If noone has won, it's a Cat's game if all of the played bits are set: 0777 (octal)

# Tic-Tac State– First solved at 0:09

## Solution

1. The bit indicating the next player is a distractor. Ignore it.
2. positions played by X (`xplay`) is `state` $>>$ 9.
3. positions played by O (`oplay`) is $\sim$ (`state` $>>$ 9)&`state`.
4. X won if for any winning bitmask (`wmask`) `xplay&wmask` $==$ `wmask`.
5. O won if for any winning bitmask (`wmask`) `oplay&wmask` $==$ `wmask`.
6. Cat's if `state`&0777 $==$ 0777.
7. If no winner and not Cat's, the game is in progress.

# MrCodeFormatGrader – First solved at 0:10

## Problem

- Given an ordered set of line numbers between 1 and N, output a compressed list of this set of numbers and a compressed list of its complement

## Solution

- Create an array of "start/stop" pairs, one pair for each segment of consecutive line numbers in the input
- Form an output string of either single line numbers (start == stop) or hyphen-separated pairs, separated by commas and spaces
- Replace the last comma with "and" and print error list
- Repeat the process with the complement of the input set to get the compressed list of correct lines

# Pizza Party! – First solved at 0:18

## Problem

- Given a set of logical constraints on pizza toppings, determine the size of a minimal set of toppings that satisfies the constraints

# Pizza Party! – First solved at 0:18

## Solution

- Divide input into three categories: *absolute*, *and-conditional*, *or-conditional*
- For every implicative, save the chain of antecedents as well as the consequent (storing as a tuple helps). Using a set for the antecedents simplifies any repetition.
- Loop over the absolute topping list. While you still have any *and-conditionals* or *or-conditionals* left, check their antecedents for the absolute topping you're looking at. Remove it from the set if it exists.
- When a set of "and" antecedents empties out, add the consequent to the back of the absolute list.
- When any member of a set of "or" antecedents is found, add the consequent to the back of the absolute list.
- Once you're done checking, removes any duplicates from the absolute list and return the size of the resulting set

# Alien Numbers – First solved at 0:19

## Problem

Given integer $X$, find the closest alien integer to $X$, i.e. an integer which does not share any digit with $X$. All values are expressed in decimal notation.

## Notation

- Denote by $LD$ the leading digit of $X$ and by $N$ the length of $X$.
- Denote by $DNUX$ the set of digits not used in decimal representation of $X$.
- Denote by $minDNUX$ and by $maxDNUX$ the smallest and the biggest digit in $DNUX$, respectively.

# Alien Numbers – First solved at 0:19

## Case digit 0 in X

Case 1. Digit 0 is not in $DNUX$:

- The closest bigger alien number to $X$: If there is a digit in $DNUX$ bigger than $LD$, output the smallest digit in $DNUX$ bigger than $LD$, followed by $N-1$ copies of $minDNUX$. Otherwise output $N+1$ copies of $minDNUX$.
- The closest smaller alien number to $X$: If there is a digit in $DNUX$ smaller than $LD$, output the biggest digit in $DNUX$ smaller than $LD$, followed by $N-1$ copies of $maxDNUX$. Otherwise output $N-1$ copies of $maxDNUX$.

# Alien Numbers – First solved at 0:19

## Case digit 0 not in X

Case 2. Digit 0 is in *DNUX*:

- 0 cannot be the leading digit in the result, except for single integer 0.
- 0 is *minDNUX*, it may be also *maxDNUX*.
- Adjust the rules in Case 1. Instead of *minDNUX*, use the next smallest digit in *DNUX*, where appropriate.

## No solution

If above rules fail to produce decimal representation of an integer, the corresponding closest alien number does not exist.

# Stacking Up – First solved at 0:27

## Problem

A stack has three operations:

- Push 1 onto the stack
- Duplicate the top element
- Add the two top elements, and decrement by 1 all other elements.

Given a desired stack of numbers, devise a set of stack instructions to create that stack, starting with an empty stack. The list of instructions must not be "too long."

# Stacking Up – First solved at 0:27

## Solution

- Use a simple recursion to find instructions to build a single number on the stack. If $s(n)$ is a set of instructions to generate $n$, then:
  - $s(1) =$ "1"
  - $s(n) = s(n/2) +$ "d+" if $n$ is even
  - $s(n) = s(\frac{n-1}{2}) +$ "d+1+" if $n$ is odd
- For each desired number $n$ on the stack, find instructions to build $n + k$ where $k$ is the number of "+" operations needed to build all the numbers above it on the stack. (This is easiest if you find these instructions for the numbers in top-down order.)
- Then concatenate these instructions. Since the length of $s(n)$ is $\Theta(\log n)$, the total size of the instructions fits within the given length bound.

# Sword Counting – First solved at 0:48

## Problem

- Given: An undirected graph G(V, E) with $1 \leq |V|, |E| \leq 10^5$
- Count the number of sword subtrees
- A set of 6 distinct vertices $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ form a sword subtree if:

$$\{v_1, v_2, v_3, v_4, v_5, v_6\} \in V \text{ and } \{v_1v_2, v_2v_3, v_2v_4, v_2v_5, v_5v_6\} \in E$$

- Two sword subtrees are different if they differ in at least one edge

## Naïve Solution

- Try all possible combinations selecting 6 vertices.
- This solution will run in $O(N^6)$ which can not pass the time limit.
- The size of the graph enforces a sub-quadratic solution.

# Sword Counting – First solved at 0:48

## Some Observations

- Every sword subtree needs to have a vertex $p$ with the degree of 4. So the degree of vertex $p$ in the original graph must be at least 4.
- Every sword subtree also needs to have a vertex $q$ with the degree of 2. So the degree of vertex $q$ in the original graph must be at least 2. Such vertex $q$ will need to be connected to a vertex $p$ where $degree(p) \geq 4$.

## Tree Version

- First let's solve the problem for the case where G is a tree. We will build our final solution on top of this simple solution.
- We can iterate over all vertices $p$ which satisfy $degree(p) \geq 4$ and fix this vertex.
- Then we can iterate and fix all vertices $q$ which are adjacent to $p$ and also $deg(q) \geq 2$.

# Sword Counting – First solved at 0:48

## Tree Version

- First let's solve the problem for the case where G is a tree. We will build our final solution on top of this simple solution.
- We can iterate over all vertices $p$ which satisfy $degree(p) \geq 4$ and fix this vertex.
- Then we can iterate and fix all vertices $q$ which are adjacent to $p$ and also $deg(q) \geq 2$.
- It can be shown that if there are no cycles in the graph, the answer will be:

$$\sum_{p} \sum_{q \in adj(p)} \binom{degree(p) - 1}{3} * (degree(q) - 1)$$

- Time complexity: $O(N + M)$

# Sword Counting – First solved at 0:48

## Graph Version

- It can be shown that the algorithm presented for the trees can also work for any undirected graph which does not contain cycles of length 3 (a.k.a triangles).
- If the given graph contains triangles, we can first count the number of sword subtrees using the previously discussed algorithm. Then we deduct the invalid subtrees from the counted subtrees.
- If three vertices $v_1, v_2, v_3$ form a triangle, then the number of invalid subtrees generated by this triangle is:

$$2 * (\binom{deg(v_1 - 2)}{2} + \binom{deg(v_2 - 2)}{2} + \binom{deg(v_3 - 2)}{2})$$

- By finding all of the triangles in the given graph, we can calculate the number of sword subtrees.

# Sword Counting – First solved at 0:48

## Some Observations

- We call a vertex $u$ heavy if $degree(u) \geq \sqrt{|V|}$. Otherwise we call it a light vertex.
- The number of heavy vertices in a graph is $O(\sqrt{|E|})$.

## Graph Version

- Each triangle either contains no light vertex or it contains at least one light vertex.
- In order to detect all triangles with no light vertices: Choose any 3 combinations of heavy nodes. $O(|V|\sqrt{|V|})$
- In order to detect all triangles with at least one light vertex: Fix the light vertex, then iterate over all pairs of its neighbors. $O(|V|\sqrt{|V|})$
- Connectivity of two vertices can be checked in logarithmic time.
- Overall time complexity: $O((N\sqrt{N}\log N)$ *where* $N = \max(|V|, |E|)$

# Avoiding Asteroids – First solved at 0:56

## Problem

Given a path of a starship (line segment) and set of asteroid directions (rays), determine if any asteroid could collide with the spaceship.

# Avoiding Asteroids – First solved at 0:56

## Key Insights

- The tests are conservative: testing whether you possibly could intersect, so use maximum bounds.
- Since you don't know how the asteroids will rotate, can treat them as spheres. Radius is maximum distance of a point on the convex hull to the center of mass.
- Asteroid danger paths are therefore semi-infinite cylinders, capped by a hemisphere.
- If an asteroid danger path overlaps with any part of the spaceshp path, there could be a collision

# Avoiding Asteroids – First solved at 0:56

## Solution

- Calculate the shortest distance from each asteroid path to the spaceship path
- If this distance is less than the asteroid radius, there could be a collision

## Shortest Distance

- Could calculate shortest distance between infinite lines. Represent lines parametrically and find parameters of point of closest approach.
- Could also use a ternary search of the asteroid and spaceship paths to find the minimum distance between line segments (initializing asteroid path to a very long line segment that extends beyond the bounds).

# Avoiding Asteroids – First solved at 0:56

## Special Cases to Handle

- Asteroid path and spaceship path are parallel
- Asteroid's initial posiiton could hit the spaceship
- Asteroid's path overlaps the line of the spaceship, but either before the starting point or after the base.
- Asteroid's closest point of approach to line of spaceship would not intersect spaceship (e.g. it is behind the spaceship), but the asteroid would hit it earlier in its path.

# iLove Strings– First solved at 1:09

## Problem

- An "`ilove`" string is a sequence of 5 distinct (distinguishing between upper and lower case) characters that start as a vowel and alternate between vowels and consonants.
- The "`loveliness`" of a string is the number of subsequences that form a (not necessarily distinct) "`ilove`" string.
- Given a string of up to $10^5$ characters determine the string's "`loveliness`" mod $10^9 + 7$.

# iLove Strings– First solved at 1:09

## Problem

- An "`ilove`" string is a sequence of 5 distinct (distinguishing between upper and lower case) characters that start as a vowel and alternate between vowels and consonants.
- The "`loveliness`" of a string is the number of subsequences that form a (not necessarily distinct) "`ilove`" string.
- Given a string of up to $10^5$ characters determine the string's "`loveliness`" under mod $10^9 + 7$.

## Naïve Approach

- 5 nested loops could be used to try all possible subsequences.
- A counter could be incremented everytime an "`ilove`" sequence is found.
- It would take $\mathcal{O}(|S|^5)$ time, which would be too slow.

# iLove Strings– First solved at 1:09

## DP Approach 1

- Build the string from some prefix subsequence.
- Keep track of which letters have been used and position in $S$.
- Try all possible next letters, and skip, if prefix is invalid.
- It would take $\mathcal{O}(|S|^2|\Sigma - V|^2|V|^2)$ time, where $\Sigma$ is the set of letters, and $V$ is the set of vowels. This is still too slow.

# iLove Strings– First solved at 1:09

## DP Approach 2

- A DP table can be kept that stores the counts of the used letters of prefix subsequences.
- For each letter in $S$ we update the table counts and the answer if applicable.
- It would take $\mathcal{O}(|S||\Sigma - V|^2|V|^2)$ time, where $V$ is the set of vowels, which is very close to the correct runtime, but still a little bit too slow.

# iLove Strings– First solved at 1:09

## DP Optimizations

- A meet in the middle approach can be used.
- Two passes one forwards and one backwards build up a prefix and suffix subsequence.
- One pass will build up 3 letters and the other will build up 2 letters.
- One table can use inclusion exclusion to quickly sum the valid spots.
- This would take $\mathcal{O}(|S||\Sigma - V||V|)$ time, which is the intended runtime.

## Beware of Memory Consumption

Using too much memory can cause TLEs. To avoid this once the final table of the first pass is constructed the table can be modified to previous states when passing through $S$ in the opposite direction. $\mathcal{O}(|S| + |\Sigma - V||V|)$ memory should be used.

# Dimensional Analysis – First solved at 1:36

## Problem

Given a set of equations involving only products/quotients, determine if any variable must be equal to 1.

## Key Ideas

1. Take the logarithm of all equations to turn them into a system of *linear* equations
2. Put the system of equations into row-reduced echelon form (RREF)
3. Reformulate the problem statement as a condition on the pattern of zeros and nonzeros in the RREF

# Dimensional Analysis – First solved at 1:36

## Conversion to Linear Equations

Every input can be converted into a homogeneous linear equation $A\mathbf{x} = 0$, where $\mathbf{x}$ contains logarithms of the physical quantities:

## Example

$$\begin{aligned} F \cdot B = X \quad &\Rightarrow \quad \log F + \log B = \log X \\ F = X \cdot B \quad &\qquad\qquad \log F = \log X + \log B \end{aligned}$$

or in matrix form,

$$\begin{bmatrix} 1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} \log F \\ \log B \\ \log X \end{bmatrix} = 0$$

# Dimensional Analysis – First solved at 1:36

## Reducing the Matrix

Put the matrix $A$ in row-reduced echelon form. For example:

$$\begin{bmatrix} 1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

## Analysis

- If a row contains *exactly one nonzero* (which must be a 1 in some column $i$), we have proved that $\log X_i = 0$ for the $i$th quantity $X_i$. So $X_i = 1$ and equations are invalid.
- Converse is also true: if some algebraic manipulations can prove $X_i = 1$, then some sequence of row operations result in a row that encodes $\log X_i = 0$. So: if no row contains exactly one nonzero, no quantities can be isolated from the others, and original equations are valid.

# Dimensional Analysis – First solved at 1:36

## Implementation Issue

Naive Guassian elimination over $\mathbb{R}$ can result in precision issues. (For some test cases, entries in the RREF matrix can become as small as $2^{-100}$).

## Solutions

1. Perform exact Gaussian elimination over $\mathbb{Q}$
2. Use a probabilistic approach: do Gaussian elimination modulo several large random primes

# Dimensional Analysis – First solved at 1:36

## Rational Gaussian Elimination

- Can prove: the entries that appear during Gaussian elimination are always the ratio of two different minors of the original matrix
- If kept in lowest terms, entries during Gaussian elimination are thus fractions $\frac{a}{b}$ where $a, b$ have at most $\sim 400$ digits.
- Implement Gaussian elimination using BigInteger fractions (in Python or Java ideally)

# Dimensional Analysis – First solved at 1:36

## Probabilistic Approach

- Solving the problem modulo a prime $p$ gives the correct answer if none of the matrix minors are divisible by $p$
- Try multiple large primes $p$ and vote to get the right answer with high probability
- Alternatively, can deterministically get right answer by trying with primes $p_1, \ldots, p_k$ with $p_1 \cdot p_2 \cdots p_k > \det A$, and carefully tracking which quantities each prime certifies *cannot* be equal to 1.

# Venn Intervals – First solved at 2:46

## Problem

Given a list of regions describing overlapping intervals, find non-degenerate placement of those intervals.

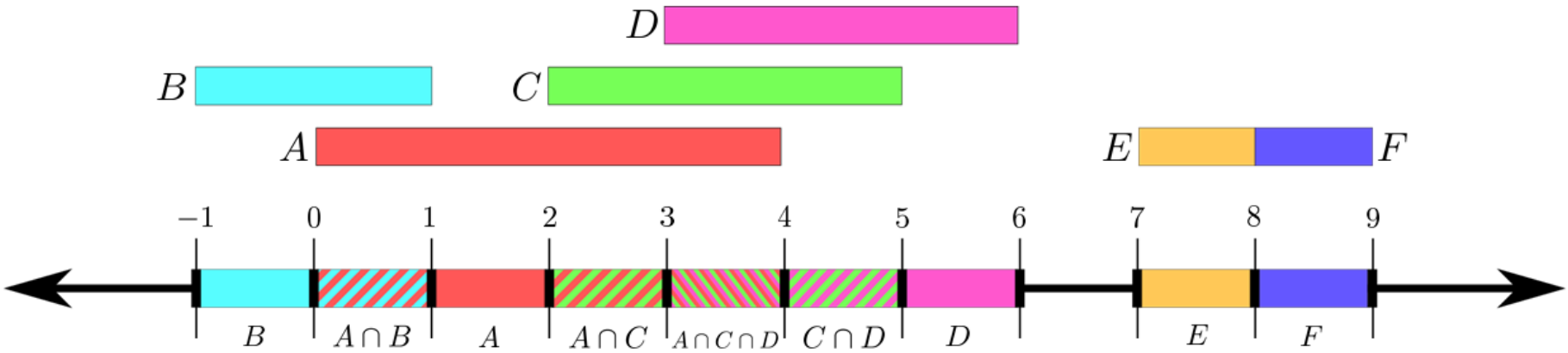
## Key Idea

Non-degeneracy condition allows a greedy algorithm. Split the regions into connected components, and build each component by placing intervals from left to right.

## Alternate Solution

*PQ*-tree data structure can be used to solve many overlapping-interval problems, and can solve this problem too. But complicated to implement.

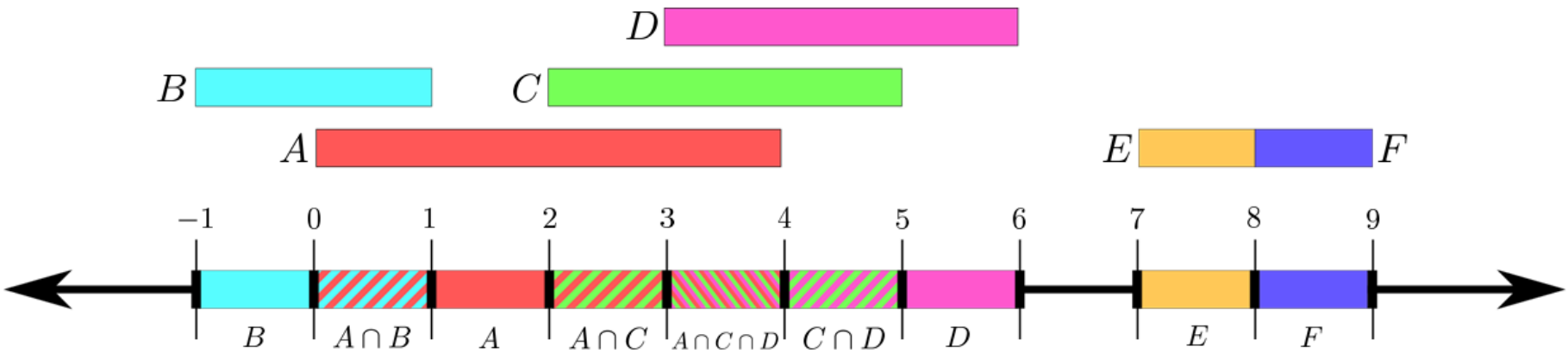
# Venn Intervals – First solved at 2:46




## Some Consequence of Non-Degeneracy

- No two intervals can start at the same place, or end at the same place
- Scanning the number line from left to right: each region is one "edit distance" away from its next and previous region
- If we know the region at the beginning of an interval, we can uniquely determine the sequence of other regions containing that interval For example, after $A \cap B$ must come $A$, become no other region contains both $A$ and $B$. After $A$ must come $C$, because all remaining regions that contain $A$ also contain $C$.

# Venn Intervals – First solved at 2:46




## Connected Component Endpoints

If an interval $I$ is the first in a connected component of regions:

- The lone interval $I$ must be a region
- There must be a region $I \cap J$ for *at most* one other interval $J$.

These two properties characterize the connected component endpoints.

# Venn Intervals – First solved at 2:46

## Greedy Algorithm

1. Set `current-pos` to be 0
2. Split regions into connected components
3. For each component, build the intervals inside it:
   1. Find one endpoint (doesn't matter which): interval $I$ where $I$ is an input region and $I \cap J$ is an input region for at most one $J$.
   2. Set `current-region` to be $I$
   3. While not done with the connected component:
      1. If `current-region` contains an interval that never appears in any remaining input region, remove that interval from `current-region`
      2. If `current-region` is an input region, remove it from the list of input regions and increment `current-pos`
      3. Otherwise, find any interval $J$ so that `current-region` $\cap J$ is an input region. Add $J$ to `current-region`.
4. During the above, remember for each interval the `current-pos` when you open it and close it. Print these endpoints.

# Venn Intervals – First solved at 2:46

## Additional Details

- If at any point you get stuck, or if you have leftover regions at the end of the above procedure, the problem is `IMPOSSIBLE`.
- With careful use of hash tables etc. the whole greedy algorithm is $O(n^2)$.
- Can use BitSets to represent regions, for a constant factor 1/64 speedup. $O(n^3)$ with a 1/64 constant factor is also fast enough to pass.