2020 ICPC North America Championship Programming Contest

Problem Solution Outlines

Another Coin Weighing Problem

- You have some bags of coins.
  - Each bag contains exactly coins.
  - Exactly one bag contains only counterfeit coins
    - Call this the fake bag
  - All other bags contain only real coins

- You have a scale
  - You can use the scale at most times
  - Right side and left side
  - Place an equal number of coins on each side
  - Scale reads a real number
    - Difference in weight between the two sides in grams

- Given and , what's the largest number of bags for which you can determine the fake bag?
  - You must specify beforehand all weighings you want to perform
    - you cannot adjust what gets weighed in future trials based on the results of previous trials

- Let real coins weigh , and fake coins weigh .

- Since we put an equal number of coins on each side at each weighing, the results of the weighings will be for some integers
  - We don't know in this case, but we can scale all the results so that they are relatively prime integers.
  - Each unique result then can be mapped to an original bag, thus the problem reduces to counting the number of valid relatively prime tuples.

- We can map a tuple , with onto each bag.
  - In the weighing, we will put coins on the left side if , otherwise, we will put coins on the right side.
  - Note we need to handle cases where some can be zero, but this can be done by fixing the number of zeros and doing some binomial coefficients.

- To count this, we can use the mobius inversion formula to do inclusion/exclusion.

- Time complexity is .

- Given a small game of battleship (at most) and the hits and misses on one player's board, determine the number of possible opponent's ship placements.
  - Ships are placed horizontally or vertically
  - Ships are distinct, even if they have the same size
  - Orientation isn't important, only squares covered

|			
	0	**X**	
0			**X**

0 = Hit!
X = Miss

- It's a simple recursive backtracking problem, recursing on placing the ships
  - At each level, place one ship:
    - For each way to place the ship in the remaining grid:
      - Place the ship
      - Recurse to the next ship
      - Unplace the ship
  - If you can recurse down to where all ships are placed, that's one placement to count.
  - The most placements is for an empty 55 grid with five size-two ships
    - That totals 18,840,000 placements, which is doable one by one.
    - No need for fancy combining algorithms

- But, there are two traps:
  - First, you must place each ship horizontally, and then vertically, UNLESS it's a ship of size one. Then, Vertical and Horizontal placements are the same
  - Next, once you've placed all the ships, you must check that there are no leftover hits ('O')

Bomas

- A zoo has enclosed area formed by circular fences (Bomas)
  - The Bomas do not intersect or touch but they may nest

- Animal types must be separated by an empty area
  - For any two area that share a border fence, at most one may house animals

- The zoo wants to add a new Boma.
  - There will be multiple queries
  - For each, what's the most areas where animals can be placed?
  - The area outside all bomas can house animals
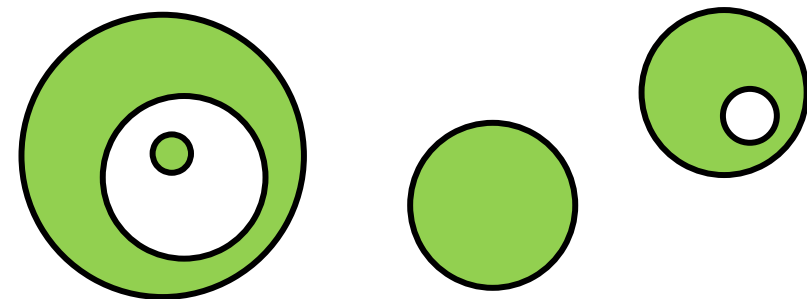
- Computing the inclusion hierarchy
  - Vertical sweep line, process events from left to right
  - Each circle has two events: open (on its far left) and close (on its far right)
  - An open event inserts two objects to track: the upper and the lower semicircles
  - A close event removes those semicircles
  - It's guaranteed that no circles intersect, so the ordering of these semicircles is stable for all positions of the sweep line
    - Caveat: enforce that the upper semicircle of circle is always above the lower semicircle of circle , since they intersect each other at the moment of insertion
  - events, objects, total processing time using balanced BST for sweep line

- Single query, inclusion hierarchy known
  - Tree DP
    - Max number of chosen circles inside circle given that circle itself chosen
    - Computable based on that of 's immediate children
  - Except this is actually greedy
    - Always choose the innermost/leaf circles
    - If the max number for a child of is achieved by choosing that child, the max number for when choosing cannot be higher than the max number for when not choosing , so not choosing is strictly better because it grants the possibility of choosing 's parent

- Multiple queries, inclusion hierarchy known
  - Include the query circles in the tree of circles
  - Propagate the information up through query circles appropriately, so that the real circles have their values and selection set as though the query circles didn't exist
    - "Choose" a query circle if any of its immediate children are chosen
    - "Value" a query circle as the sum of its children, NEVER adding 1 for the query circle itself
  - When reporting back the value of each query after computing the values for the entire tree, add 1 as needed to account for the query circle itself (i.e., if it is not "chosen")

All Kill

- You and a friend want to All Kill a programming contest
  - That means to solve all of the problems
- Solving a problem has two phases
  - A thinking phase and a coding phase.
  - Your friend is responsible for all the thinking while you are responsible for all the coding.
- You use this strategy:
  - For each problem that doesn't yet have an idea, your friend will get the idea to solve it with probability 1/(number of minutes remaining).
    - Your friend can get the idea to solve multiple problems in the same minute.
  - Among the problems that still need code time and your friend has gotten the solution idea, you will take the lowest numbered one and spend the next minute coding it
  - If no problem satisfies the condition, you do nothing at this step.
- What is the probability that:
  - Your team finishes coding all the problems by the end of the contest
  - For each problem, the time spent coding that problem is a contiguous interval
- Let be this probability, be the number of problems in the contest and be the number of minutes in the contest.
  - It can be shown that is an integer. Output

- Instead of problems and time, it may be easier to think about this as placing some blocks on a physical location in a line.
  - We go through the blocks from to , and choose a number randomly, and try to place the block starting at the first empty position after the randomly chosen number.
  - If this process succeeds for all blocks, then this corresponds to a valid scenario from the original problem.
  - This process can fail if there isn't enough space for the block at the first empty position, or if there are no empty positions.

- To solve this problem, there are a few observations.
  - First, instead of placing blocks in a line with n empty spaces, let's place blocks in a circle with empty spaces.
  - Blocks will scan from a position clockwise around the circle until they find an empty space to try to go in.
  - This time, we are guaranteed every block will eventually find an empty space, but this process can still fail if there isn't enough space.
  - In the end, we can multiply the answer by , since everything in on a circle, everything is symmetric, so the count of configurations where a particular space is empty we added is the same for all spaces.

- So now, the main issue is counting the number of valid ways to place the blocks in a circle.

- As a first step, it doesn't matter where we put our first block since it's on a circle. As another simplifying assumption, let's assume there will only be one empty space at the end (we will talk about how to adjust the logic later to account for multiple empty spaces).

- After we place this first block, we get a line, and let's partition this line into n sections (we don't fix the sizes of each sections yet, this will be determined by the order of the blocks after we're done). We will imagine these sections to exactly hold one block, and one section will be left empty.

- For the second block, we choose a spot where it starts searching. There are two cases, this spot is occupied or empty. If the spot is empty, this means it must be the beginning of some section, so there are n ways to choose this. If the spot is occupied, then this second block will end up in a section that immediately follows an already occupied section. There are ways to choose this spot.

- For the third block, we choose a spot where it starts searching. Again, there are two cases, this spot is occupied or empty. If this spot is empty, this is the beginning of some section, of which there are n-1 ways to choose this. If the spot is occupied, then the third block will end up in a section that immediately follows an already occupied section. There are ways to choose this spot (and notice, this is always the same regardless of where the first and second blocks are adjacent or not).

- Generalizing, we can see the block has

  choices of a spot. Thus, the answer is just the product of all these choices.

- To deal with empty spaces, instead of starting with sections, we start with sections, and the resulting logic is similar.

- To summarize, the main observations are:
  - Do this on a circle instead of a line
  - Split the circle into sections, but don't fix the sizes of the sections while the process is going. The section sizes will be determined after the order of the blocks is determined

- Alice has an grid and a block.
  - She would like to place her block in the grid so that the block is axis-aligned and covers exactly 4 grid cells.

- Bob wants to prevent Alice from doing that, so he places obstacles in some of the grid cells.

- How many ways can Bob place the minimum number of obstacles to prevent Alice from placing her block?
  - Note: The answer is not the minimum number of obstacles, it's the number of ways of placing the minimum number of obstacles

- if and are odd, the answer is .

- if is odd and is even, then answer is
  - (and the case for if is even and is odd is similar)

- if and are even, we can do a bit mask dynamic programming solution.

- For every and , there exists exactly obstacle in one of the squares:

  - Call these the "aligned" subgrids.
  - We can't do any less, since otherwise, Alice would be able to place her block in this space, so this minimizes the number of blocks needed. We can use this fact to do a bitmask dp.

- Imagine we place a block in the upper left corner of each aligned subgrid.
  - We can optionally push the blocks in some prefix of each row right.
  - Similarly, we can optionally push the blocks in some prefix of each column down.
  - This will ensure that it's impossible for Alice to place her block when one of the coordinates of the top left square is even.
  - The only tricky case is to make sure we also exclude cases where both coordinates of the top left square are odd.

- We can see in this example we can run into a bad case (here, the first row has been pushed right, and the second column has been pushed down).

```
.X|..
..|.X
-----
X.|..
..|X.
```

- Fortunately, this is a local condition that we can make sure to exclude, so we can do a bitmask dp to solve this. (here, the state is length of prefix of row that's pushed right, and set of columns which we can still push down). This is still slightly too slow, so to speed it up, we can use fast Walsh Hadamard transform. The overall runtime is

Hopscotch

- There's a new art installation in town
  - An grid of tiles
  - Each tile has a number from to

- You want to play Hopscotch on it
  - Start at any tile numbered , hop to a tile numbered , then , and so on, to .

- What's the shortest possible distance?
  - Use Manhattan Distance between tiles:

- It's just a shortest path problem, so we'll use Dijkstra's algorithm

- First, a simplification:
  - Create an of lists of points, where is a list of coordinates of tiles with the value .
  - That's just a single pass through the matrix. You can even do this while reading in the matrix
  - This will keep us from having to search the whole matrix at every step of Dijkstra's algorithm

- Then, it's just Dijkstra's
  - Start by populating the priority queue with the coordinates of all tiles, with
  - Then, when taking a tile with value off of the priority queue, just use to find the next tiles
  - Of course, stop when you first reach a tile with value

ICPC Camp

- Scheduling problems to solve on each of days of Programming Contest camp
  - There are traditional problems and creative problems
    - Schedule one of each on each day
    - Cannot use any problem more than once
    - Each problem (both kinds) has a difficulty expressed as an integer .
    - The sum of the difficulties of the two problems on any given day cannot exceed some constant
    - Define a value such that the difference in difficulties on every day is less than or equal to
    - Find the smallest possible value

- A Matching Subproblem
  - If we are given and , can we determine the maximum cardinality of the matching?
  - Yes. We can use a greedy algorithm:
    - For every max element remaining in , we try to find the max element in that can be paired with it.
    - We may prove the correctness of the algorithm by showing that the greedily chosen pair always lead to an optimal answer, otherwise we can make swaps.
  - We can use some binary-searchable data structure to efficiently find the max element in . Thus the greedy algorithm takes time.

- Finding the Minimum
  - With the greedy algorithm we can binary search the minimum that gives us a matching with a cardinality of at least n.
  - Then entire algorithm runs in time, where .

- There are three horizontal wheels of letters stacked one on top of the other, all with the same number of columns.
  - All wheels have one letter, either **A**, **B**, or **C**, in each of its columns on the edge of the wheel.
  - You may rotate the wheels to adjust the positions of the letters.
    - In a single *rotation*,you can rotate any single wheel to the right or to the left by one column.
  - The wheels are round, of course, so the first column and last column are adjacent.

- What's the fewest possible *rotations*
  - so that every column has three distinct letters?

ABCCBAAC
BABBCCAC
CACACBBAB

- **Call the three wheels , and**
- **The Simple Solution:**

```
For (delta_ab in 0..n-1)
   If (good(a, b, delta_ab))
      For (delta_bc in 0..n-1)
         If (good(b, c, delta_bc) && good(a, c, delta_ab + delta_bc)
            Result min= bestcost(delta_ab, delta_bc)
```

- **In other words, Try all deltas between and , and all deltas between and , find the smallest combo that works.**

- Best cost given a pair of deltas
  - Given rotation and rotation, the rotation is just their sum (modulo n).
    - Define as

  - If we find a solution where the a ⊐ b rotation is , and the b ⊐ c rotation is , what are the best three values , , and such that

    , = - , and (all mod n)

    that minimizes ?

- This will always reach a minimum with one of the , , values equal to zero.

- Why? Consider the directions you turn the wheels.

  - If you turn all three one direction, you can improve it by turning all three one fewer twist in that direction.

  - If you turn two in one direction, you can improve it by turning those two one fewer twist in that direction and turning the third one more twist in the other.

  - Thus, the optimal value will always occur with one wheel unturned and the other two turning in opposite directions.

- This is .

- For the simple algorithm, our runtime is O(n^3)
  - The a and b wheels can be consistent O(n) times.
  - We need to try O(n) offsets for d_bc
  - Checking the c wheel against a and b takes another factor of O(n).

- At n=5000 this is too slow; we need to trim a factor of O(n).

- Given a d_ab, we can calculate what c needs to look like ignoring rotation (call it c') in O(n)
  - We can do this as we check a against b for consistency.
  - So we want to ask the question: what rotations of c (if any) are consistent with this?

- A simple solution is to just build a hash table that maps all rotations of c to a vector of rotation values.

- Another solution is to use a cyclic hash and check for a matching hash value.

- Either one of these tricks reduces the runtime to O(n^2) and passes easily.

- An even faster solution is to an Fast Fourier Transform or Number Theoretic Transform to construct a solution with convolutions. One of the judge solutions exhibits this technique, but it was not necessary to get a solution that runs in time.

Editing Explosion

- Given a string and an integer , how many distinct strings are a Levenshtein Distance of exactly from ?

- The Levenshtein Distance between two strings is the smallest number of simple one-letter operations needed to change one string to the other.
  - Adding a letter anywhere in the string
  - Removing a letter from anywhere in the string
  - Changing any letter in the string to any other letter in the alphabet

- Standard Edit Distance with Dynamic Programming
  - Given string a with length and with length

  - Build a two-dimensional array dp of size , with the rows associated with the cursor positions in string and the columns with the cursor positions in string . The entry `dp[i][j]` gives the edit distance between the i character prefix of b and the j character prefix of a.

  - Edge conditions: `dp[0][j] = j; dp[i][0] = i`

  - Otherwise, `dp[i][j] = min(dp[i-1][j]+1,      dp[j][i-1]+1, d[i-1][j-1]+(b[i-1]==a[j-1] ? 0 : 1)`

- Deconstructing the Algorithm
  - Row **i** depends only on row **i-1**, the string **a**, and (for i>0) the character **b[i-1]**.

  - Adjacent elements in a row differ by a maximum value of 1.

  - The length **a** is short; less than or equal to 10.

  - The maximum allowed distance is at most 10. (We can treat any values more than 10 as 11).

  - Thus, the number of distinct possible rows in the dp matrix, across all strings, is small.

Editing Explosion (3 of 4)

- Generalizing from one string to all strings
  - Instead of considering a single string and building each row one by one, we consider all strings of a given length, and build a hashmap mapping the row value to the count of strings that end up generating that row.

  - The initial hashmap is just representing the empty string.

  - We know the length of a matching string cannot be more than , so this sets a bound on how far to iterate.

- Iterate from 0 to d.

    - Given such a hashmap, we can iterate over it, consider all possible 26 characters to extend , and generate the new hashmap for the next length.

    - At each iteration we add the number of strings at this length that have the required edit distance value in their row at element .

Lunchtime Name Recall

- Mia needs help remembering her colleagues' names
  - She buys lunch for all of them over the course of days
  - She buys burgers and salads
    - varies per day
  - She watches her colleagues eat lunch, and notes whether they're eating a burger or a salad
  - In this way, she starts to identify her colleagues
  - Given , and values of (one for each day), what's the maximum number of colleagues that she can uniquely identify?

	A	B	C
Day 1			
Day 2	A	B	C
Day 1	1	0	0
Day 2	1	1	0

- Think of a matrix of n columns and m rows with binary values (1/0).
    - Each column represents one colleague.
    - Each day represents a row.
    - Cell (,) is 1 or 0 if we give colleague a burger or a salad on day .
    - We can reorder the values in each row.

- Consider the values in each column as a sequence of values.
    - If the sequence is unique, it means what the colleague eats over the days is distinguishable from what the other colleagues eat.
    - So we want to maximize the number of sequences that are unique, by reordering the values in every row.

- We are essentially creating partitions by distributing each day's burgers into different columns, starting with an initial partition of { }.

- For each group of size , we can assign burgers to it, and split it into two smaller groups of sizes and

Here is how we achieve answer 5 in sample 2:

```
{16}
{6, 10}                 1111110000000000
{2, 4, 6, 4}            1100001111110000
{1, 1, 1, 3, 1, 5, 1, 3}  1001111000000111
```

The groups we have are only distinguishable by their sizes. In how many ways can we split elements into groups? There are only ways, which is listed at https://oeis.org/A000041. This value is also computable by simple DP.

- is not a big number.
- We can probably track all the ways to create the partitions.
- Each partition can be viewed as a sorted list of its group sizes.
- On each day, we pick some existing partition and decide how to distribute the burgers into the groups of .
- For each group in we decide how many burgers to place in this group to split it.

- Let our state be on each day. is the number of burgers left. For the first group of size in , we enumerate , the number of burgers to put into the first group, and move on to a new state .

- Let the space of be . Multiplied by , the total number of states in each day is For each state, we enumerate choices of . Additionally to maintain and we need to modify elements in the sorted group lists after enumerating , which takes . Our cost is for each day, and in total.

- To roughly estimate , we have where is even, because the number of groups in is always even. This is about . , which seems slow. As it is a loose bound, it gives us the intuition that the actual algorithm may run faster.

There are a few places where we can get a more precise estimate:

- The enumeration of depends on the size of the first group, and also .
- cannot exceed the sum of all group sizes in .
- The cost to maintain the sorted group lists depend on how many groups we currently have.

If we run a program to take those into account precisely, we shall see that the total cost on each day is about . This provides a much better estimate on . Multiplied by , our total cost is and is feasible.

There are at least two ways to reduce the constant factor in the algorithm:

1) The number of burgers is equivalent to , we can take the min of the them, and reduce the constant by a factor of 2.
2) When we find a way to obtain the max possible answer n, we terminate immediately.

These will speed up the solution, but neither is required to solve the problem.

Rooted Subtrees

- A *Tree* is is a connected, acyclic, undirected graph
  - With nodes and edges
  - There is exactly one path between any pair of nodes

- A *Rooted Tree* is a tree with one particular node identified as the root

- A set of nodes for roots and is *obtainable* if and only if it can be expressed as the intersection of a subtree in the tree rooted at and a subtree in the tree rooted at .

- For a given pair of roots and , count the number of different non-empty obtainable sets.

- There can be multiple queries in the input file

- In the unrooted tree, consider the path between and .
- For any node not on that path, any intersection either has the whole subtree there or you don't.
- The only "interesting" nodes are the ones between and .
- You can take any contiguous sub-segment of the path between and .
- If there are nodes between and (inclusive)
  - then there are options
- To count the number of nodes between two nodes:
  - use a BIT or Segment tree to find the LCA in time, or
  - use a sparse table to find the depth of the LCA in time
- Overall runtime: or depending on implementation.

Tomb Raider

- There is a tomb (represented as an grid) with gargoyles, obstacles and mirrors
  - The walls are mirrors
  - The internal mirrors are angled at
  - The gargoyles each have 2 faces: Either top & bottom, or left & right
  - You can rotate any gargoyle by 90 degrees
  - What's the fewest number of gargoyles you must rotate so that every gargoyle face can see another gargoyle face (possibly its own)?
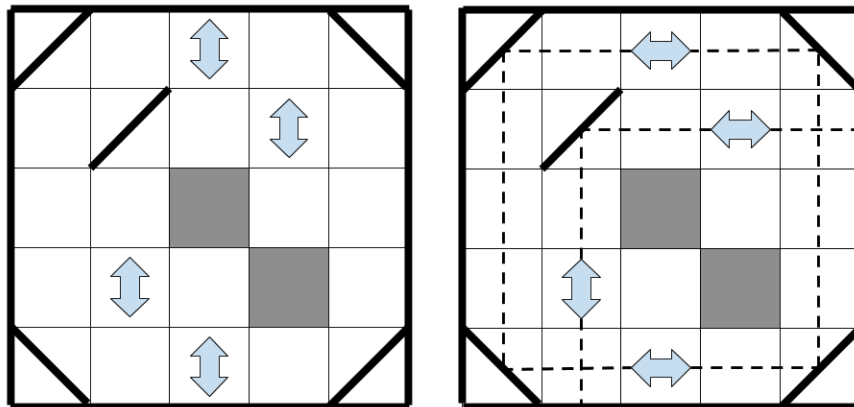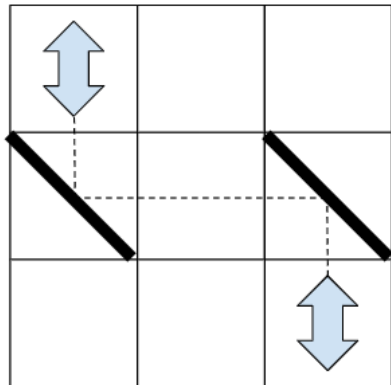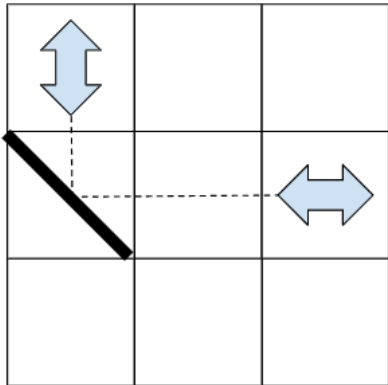

The first diagram shows a tomb.
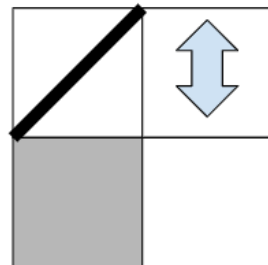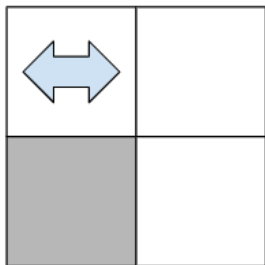The second shows a solution for this tomb, rotating 3 gargoyles

If there is a light path directly connecting two gargoyles, they must be in a same direction.

If there is a light path connecting two gargoyles via mirrors, then if is odd, the two gargoyles must be in different directions. Otherwise, they are in a same direction.

If there is a light path connecting one side of a gargoyle to an obstacle, then one direction of the gargoyle is prohibited.

- Consider **V** and **H** as black and white.
  - Then each gargoyle has a binary color and we can change its color.
  - Based on the floorplan, we want some pairs of gargoyles have the same color, some pairs have different colors.
  - How to make the smallest number of color changes so that the same/different color requirement is satisfied?

- This is a bi-coloring problem.

- Let each gargoyle be a node, and each same/different color requirement be an edge.
  - We can build the graph to bicolor.
  - We may group those nodes that must have a same color together, and treat them as one super-node.
  - After this, the remaining edges are all requirements of different colors.

- If there is an odd-length cycle in this graph, then there is a conflict, and the answer is -1.

- Otherwise, we have two choices of coloring for each connected component.
  - We can compute how many gargoyles need to be rotated for each choice, and use the one that needs fewer rotations.

- The bi-coloring graph has nodes and edges. It takes time to build it, as we only need to traverse each cell in the grid four times (from/to its four neighbors).

- Solving the bi-coloring takes time.

- Total time: