## 2025 SI@UCF CP Camp Final Contest Solution Sketches

## Problem A: Splitting Cake (Problem Author: Arup)

The first part of the problem is to find the area of the whole cake. Since we only care about ratios and need to keep things in integers, just calculate twice the area of the cake surface, which is guaranteed to be an integer. This computation is short (for loop method shown in class often called the shoelace theorem). If we were to calculate all split polygon areas from scratch, it would take too long because there are up to 100,000 different polygon splits, and those polygons can be as large as 100,000 vertices. The key idea is to recognize that the sum/difference of products for each of the different 100,000 polygons we compute are closely related to one other. Let $t(i) = x[i]*y[(i+1)\%n] - x[(i+1)\%n]*y[i]$, where $x[i]$ and $y[i]$ store the x and y coordinates of the $i^{th}$ pt (0-based), and n is the number of points in the polygon. Let $e(i) = x[i]*y[0] - x[0]*y[i]$.

For all relevant areas to the problem, we are adding terms of the form $t(i)$ and $e(i)$ only.. Let $A(i)$ represent the area of polygon i, where i is the maximum vertex number in the polygon after the cut is made from vertex 0 to vertex i. For example $A(2) = t(0) + t(1) + e(2)$. More generally, we have:

$A(i) = t(0) + t(1) + … + t(i-1) + e(i)$

It follows that

$A(i+1) = A(i) - e(i) + t(i) + e(i+1)$.

In short, we can compute each of the areas of the "next" polygon cut using the answer from the previous polygon in O(1) time by utilizing the formula above.

One final note: Depending on the order of the vertices (clockwise vs counter-clockwise), the area calculations could be positive or negative. It's probably safer to always take the absolute value at the very end of each area calculation so that the standard gcd code works. The very last step is to calculate double the area (guaranteed to be an integer) of both polygons for each cut, take the smaller of the two, and try to maximize this value. Right before printing, divide both values by the gcd of the two values so that the output is two relatively prime integers.

## Problem B: Fake Writing (Problem Author: Brygida)

To solve this problem, it helps to abstract away the vowel/consonant aspect and instead imagine both strings as binary strings, where each vowel is represented as a 1 and each consonant is represented as a 0. By definition, in order for a substring to be valid, it must have the same ratio of 1's to 0's as the target string. We can represent this ratio as a fraction, **a/b** where **a** is the number of 1's in the string and **b** is the total number of digits in the string. We can then simplify this fraction by finding the greatest common divisor of both the numerator and the denominator, and then dividing both by the greatest common divisor. The resulting fraction is **c/d**. Any substring of the notes that is valid must have a total of **c** * n 1's and **d** * n digits total, where n is any non-zero positive integer. Therefore, when determining the "representative fraction" of any substring (the number of 1's divided by the length of the substring), we can think of every digit as always adding 1 to the denominator, while 1's add 1 to the denominator and 1 to the numerator.

Since all digits contribute equally to the denominator, the denominator is easy to determine given the length of the string (in this problem, the denominator is equal to the length of the string). We can then represent our initial binary string of 1's and 0's as an array of integer values, each 1 or 0, each representing how much the digit at the index they occupy contributes to the numerator of any string containing it. To make the solution simpler, we can do some algebra on this array. We can multiply all values in our array by **d** to turn it into an array containing only **d**'s and 0's, while multiplying the target fraction **c/d** by **d** converts the fraction to **c**. We can then subtract **c** from both sides, to make our array contain only (**d** - **c**) and -**c** while our target becomes 0. Since we only performed algebraically valid operations on either side of the equation, all we need to do to find a substring that satisfies our original condition (number of 1's / length of string = **a/b**) is to find a subarray with a sum of 0.

We can do this in O(**n**) by sweeping from left to right in our array, and keeping a running sum of the values seen in it. We also keep a map that stores the total number of times any sum has been encountered. This is because if any prefix of the array has a value equal to the value of the array up to our ending point, we could remove that prefix, subtracting its value from our sum and leaving us with a subarray with a sum of 0. Note that this means our map should start with one way to get a subarray with a sum of 0, since we could remove no prefix from the string and just have the running sum be the value of our subarray. Of course, the only way this forms a valid substring is if our running sum is 0.

## Problem C: Good Groupings (Problem Author: Justin)

This problem can easily be solved with the use of sets. Keep a set of TAs. When reading in the group suggestions, use the set to check if any of the members are TAs. Also, make sure the group size is 3 or greater. That's it!

## Problem D: One Last Ride (Problem Author: Justin)

The first step to solving this problem is to realize the graph you are given is a tree (the problem description intentionally obfuscates this). From here, we should root the tree at an arbitrary node and determine the parents of each node.

For each query, you are given 2 nodes **a** and **b** and you must find the simple path between them. Define node **c** as the lowest common ancestor of **a** and **b**. The path from **a** to **b** can be split into two paths: traverse up the tree from **a** to **c**, followed by descending down the tree from **c** to **b**. So, all the nodes that you will visit will be all the direct ancestors of **a** until **c**, and all the direct ancestors of **b** until **c**.

It is not difficult to do this in linear time with the use of a DFS or BFS. However, to bring the time per query down to logarithmic, we should use *binary lifting* (learn that first before solving this problem) to precompute the ancestors of each node that are a direct power of 2 above them, and with it, keeping track of the sum of all edges, as well as the minimum node along each jump upward (kind of similar to the structure of a sparse table). This allows us to binary search for the **LCA(a, b)**. When traversing up the tree, the sum of the edges and the minimum node along all paths upward with lengths equal to a direct power of 2 are precomputed, so it is no longer needed to visit each node individually. Keep track of a running sum of edges, as well as a minimum and update them accordingly.

Finally, use the computed sum of edges and the minimum vertex on the path to case out which statement to print. If the sum of the edges is greater than the query time, you will be late no matter what. If the sum of the edges is less than or equal to the query time, it is possible to get back in time, but to determine if you can go on a ride, the sum of the edges plus the minimum vertex on the path must be less than or equal to the query time.

## Problem E: An Optimal Lunch (Problem Author: Justin)

To answer the queries, you have to answer questions of the form "What's the shortest distance from s to x?", where s is the start location and x is a potential lunch location, and questions of the form, "What's the shortest distance from x to e?" where x is the potential lunch location and e is the ending location. Since there isn't a single source to all of the queries, it's best to simply pre-compute the shortest path between all pairs of vertices. The standard algorithm to do this is Floyd-Warshall's which runs in $O(V^3)$ time for a graph with V vertices. For this problem, $V \leq 300$, so this algorithm runs fast enough. ($300^3 < 10^8$).

One other slight issue complicating the solution is that the queries are NOT given by vertex number, but by the name of the rides. Thus, a map must store a look up table from ride name to ride number to ensure the code runs fast enough.

## Problem F: Palindromer Quest at Universal (Problem Author: Christian)

There are two types of palindromes: even lengthed ones and odd length ones. Consider all palindromes of length 5. If the start with the digits abc, then the last two digits must be b and a. There are 9 choices for a and 10 choices for both b and c, meaning that there are 900 total palindromes of 5 digits. We can label these "numbers" 100 to 999, the prefix of the palindromes. These are quite easy to count without enumerating all of them. The same is true of 6 digit palindromes - once you know the first three digits, the last three digits are set, so there are also 900 total 6 digit palindromes. Once this observation is made, no brute force has to be done. The only difficulty are the corner cases with the very first palindrome in range and the very last. Consider determining the number of palindromes in between 237 and 81860. We certainly have all 4 digit palindromes and can compute this number very quickly. What remains is determining the number of 3 digit palindromes greater than or equal to 237. The only question is whether the first one is 232 or 242. This has to be coded (reflect the number and then test if it's in range or not). This means we start with palindromes that start with 24 and end with 99, meaning that there are 99 - 24 + 1 = 76 3-digit palindromes in the given range. Similarly, we note that 81818 is within the range so the total number of 5-digit palindromes are ones that start with the digits 100 through 818, so there are 818 - 100 + 1 = 719 of these. It follows that for this example, there are a total of 76 + 90 + 719 = 885 palindromes in between 237 and 81860, inclusive. Since each part of the calculation is O(1), and there aren't more than 12 digits, solving queries can be done extremely quickly. Another potential solution would be to write a function, numPals(n) that returns the number of palindromes less than or equal to n and solve the range query via subtraction: numPals{high) - numPals(low-1).

## Problem G: Radix 67 (Problem Author: Arup)

This is a base conversion problem at the heart of it, with another layer added on top. First, read in the input and separate it into substrings of length 6. For each substring of length 6, convert this adjusted binary notation to a decimal number. (There are several ways to do this.) In general, if the character c is storing the input, the expression c - '6' will properly evaluate to 0 and 1 for the corresponding bit. Once the binary value of the string is uncovered, this has to be converted to an Ascii character. It's good to write a function that breaks this into cases: upper case letters, lower case letters, digits, '+' and '/'. Adding a base ascii value ('A' or 'a' or '0') to the appropriate numeric value will result in the correct character. (For example, if the binary value we get is b = 44, since this is in between 26 and 51, we want the character 'a' + (b - 26), since the numeric value we want is offset by 26.)

## Problem H: Seating (Problem Author: Brygida)

For this problem, you simply need to find the remainder after the students are seated into their rows, and then subtract that remainder from the total number of seats in each row. The built-in modulo operator, "%" works to find the remainder.

There is only one special case. When the number of riders that fit in one row is a divisor of the number of students, the modulo operator will return 0. Without handling this situation specially, our function will return the number of seats in each row, since we will calculate (number of seats in each row - 0). However, there will be 0 empty seats in any row occupied by students. Fortunately, we can easily check for this case by evaluating the return value of the modulo operator. Since **a** % **b** only returns 0 when **b** is a divisor of **a**, any time the modulo operator returns 0 we know we are dealing with this special case and can print 0 without further calculation.

**<u>Problem I: The Final Substring Showdown (Problem Author: Christian)</u>**
To solve this problem efficiently - especially for strings up to 1 million characters long - we need a data structure that captures all substrings of a string and how many times each one appears. A brute-force solution (e.g. generating all substrings and counting them) would be too slow. Instead, we use a Suffix Automaton (SAM), which is a powerful tool for representing all substrings of a string in compact form, while also allowing us to compute how many times each substring appears. The time complexity is O(N) per test case.

**Step-by-step Strategy**
1. Build the Suffix Automaton (SAM)
    a. A SAM is a state machine where each state represents a set of substrings that share the same right-end positions in the original string.
    b. As we iterate over the string character by character, we update the SAM incrementally, extending its transitions and linking new states to previously visited states.
    c. Each state holds the length of the longest substring it represents, a link to its suffix, and a map of transitions to next states.
2. Compute Frequency of Each Substring
    a. Initially, each end position in the original string corresponds to a unique path ending at some SAM state. So we set the count of each newly created state to 1.
    b. Then, we propagate these counts from longer substrings to shorter ones using a reverse topological order (i.e., sort states by length descending).
    c. This lets each state accumulate the total number of times its substring appears as a suffix in the string.
3. Count Distinct Substrings by Frequency
    a. For every state u (except the initial root), it represents all substrings whose lengths fall in [len(link[u])+1, len[u]]. So the number of substrings this state contributes is len[u] - len[link[u]].
    b. If the frequency of these substrings (i.e., cnt[u]) is at least k, we add this number to our answer.
    c. To efficiently do this, we use a freq_substring_counts[freq] array to accumulate the count of substrings that appear exactly freq times.
4. Answer the Query
    a. After processing all states, we sum over all frequencies f >= k and return the total number of substrings that appear at least k times.

**What makes this work?**
● The SAM compactly stores all substrings in O(N) space and time, where N = |s|.
● Each substring corresponds to a path in the automaton, and the count propagation ensures we know how many times each path is taken.
● The clever part is realizing that the number of substrings a state contributes is not 1, but proportional to the difference in lengths between it and its suffix link.

**Problem J: Cola Surfing (Problem Author: Brygida)**
This problem can be solved with a slightly modified breadth first search (BFS). In a breadth first search, you mark the distance of each reachable location from a start location. In this problem, we can think of the distance as being time steps. If there were no octopi, then you could run a BFS from the starting location to mark the time at which you would reach each square for the first time. If any of the squares in the last two rows are reached, we made it.

The question now is, to what extent do the octopi complicate things. The key observation is that at each time step, either both you and the octopi move down by two rows, so that your relative difference in rows is the same, OR the octopi gain 1 row on you. Thus, it's **IMPOSSIBLE** for the octopi to move below you without being on your row first. Thus, it's always optimal to move down as fast as possible (maximize the number of down 2 moves.) Thus, the only possible paths that could escape are the ones that escape in the fastest number of time steps, which is precisely what BFS calculates!!! So, run the BFS, and then just check to see if you reach one of the bottom two rows before the octopi do. There's one final exception: you do get out alive if the first time the octopi occupy your row is when you've already escaped. So, if the octopi first reach your row when you escape (meaning that your last move was a diagonal move to the second to last row), then you ALSO escape. Otherwise you don't! Alternatively, you can code your BFS to never enqueue a square that an octopi reaches at the same time step as you (except the bottom two rows.)

**Problem K: Top K for the Crown (Problem Author: Christian)**
It's clear that the goal is simply to find the k largest numbers in the input and add them. The easiest way to do this is use a built in sort function to sort the values, and then add up the appropriate ones. (If you sort in increasing order, then take the last k, from index n-k to index n-1. If you sort in decreasing order, then add the items in index 0 through index k-1, inclusive.) Also note that the numbers are large, so you have to use long/long long in Java/C++.

**Problem L: Uneven Walk (Problem Author: Arup)**
This was meant to be one of the easier questions in the contest, but one optimization must be made because Arup could need to walk up to $10^{18}$ inches. Instead of simulating the steps, one can note that in 2 steps Arup always walks exactly **r+s** inches. Thus, integer division can be used to compute the number of full pairs of steps he needs to take. So, for example, when **r** = 15 and **s** = 9, he walks 24 inches every 2 steps. In order to walk 86 inches, he must take 86/24 = 3 pairs of steps, which is 6 steps total. From there, he either needs to take 0, 1 or 2 more steps. In 6 steps he walked 24 x 3 = 72 inches. Since the destination is farther, he must take 1 or two more steps. Since 72 + 15 = 87, and this is greater than or equal to the 86 inches he needs to go, the final answer is 7. If this was too small, then the answer would be 8. Thus, this question was testing the use of long long/long (since the input can't fit in an int) as well as integer division, mod and some case work. (Mod is what allows us to differentiate between 0, 1 or 2 extra steps.)