

2026 SI@UCF Introduction to Competitive Programming Final Contest Solution Sketches

Problem A: Arctic Network

This is one of the most challenging problems in the set. Without knowing the radio connection radius, it's difficult to determine the number of components (satellite systems) that the graph would be split into. However, if the connection radius is known, we could calculate which pairs of locations would be connected via radio, and then use either DFS or BFS to count the number of connected components that graph has. Notice that as the radius increases, the number of components decreases. Thus, we can think of $f(r)$ as a function with the input as the radius of radio communication and the output of the function being the number of connected components (satellite systems) needed. This is a decreasing function, so we can binary search over it. Thus, the binary search is on the radius and if a radius produces too many components, then the real answer is bigger than that radius. If the radius produces less than or equal to the maximum number of components, the real answer is less than or equal to this radius. Thus, the solution requires a binary search and when evaluating a radius, either a DFS/BFS must be implemented on the corresponding implicit graph. (It's not necessary in code to actually form the graph.)

Problem B: Collecting Beepers

Since we can visit the beepers in any order and there are 10 or fewer of them, we can just try every permutation of picking up the beepers. For each permutation, we have to add up the total Manhattan distance traveled, which includes both the first path from the start position to the first beeper to collect and the last path from the last beeper to collect back to the original starting position. In C++, the next permutation function can be used, or you can write the algorithm yourself.

Problem C: Bus Assignment

This is a simulation problem. Simply keep track of how many people are on the bus at all times, processing the data in the order requested in the problem statement. Each time this number changes, see if the value is greater than the maximum seen so far. If so, update the maximum. The maximum seen at the end is the answer.

Problem D: Gluttonous George

This is the easiest problem in the set. Read in three items, an integer, a string and an integer. Compare the two integers, and output accordingly, based on the problem specification for the 3 different cases.

Problem E: Hidden Password

The valid rules are covered in the problem specification and the input is small enough that efficiency isn't necessary. When looking for the next letter in the message to match the password letter $p[i]$, you have to start where you left off in finding $p[i-1]$ (or the beginning for $p[0]$) and you need to make sure you don't see any character of the form $p[j]$ where $j > i$. We can do this by first checking to see if our current message letter is $p[i]$, if it is, we can advance to the next letter in the password. If not, we then check if the current message letter is any of $p[i+1]$, $p[i+2]$, ..., $p[n-1]$, where n is the length of the password. If it is, our answer is not valid and we should just return this (from a function) immediately. Continue in this fashion and if you mark off every password letter in the message without ever returning that it can't be done, then it passes.

Problem F: Kattis Speedrun

Due to the limitations of the input, it's clear that the total number of problems Joshua needs to solve to achieve his goal is at most 100,000 and that the optimal answer is based on him solving in between 0 and 100,000 problems total. Thus, we can brute force trying to solve each number of problems. Once we know how many problems we're solving, we can set up a math equation to solve for the minimum number of problems we need to create to get to our goal. Thus, we try at most 100,000 options, and pick the best one. Note that if the mathematical solution for the number of problems to create gives a negative answer, we must just reset it to 0.

Problem G: Mylla

There are 8 ways to win tic-tac-toe. For this problem, you just have to check all eight to see if any of them have three X's in a row. If so, output "Jebb", otherwise output "Neibb". There are many ways to check these eight possibilities. One succinct way is to store in a constant array, the set of indexes into the input that constitute a single solve line, and then loop through that constant array, using it to index the input string/strings.

Problem H: Poker Hand

We only care about the first character in each input string. We can just keep a frequency list of how many times each character appears. This can be done with an array or a map. Our output is simply the maximum frequency of any first character over the five strings.

Problem I: Ptime

Since the exam is at most 100 questions, one could use string concatenation to produce strings for Adrian, Bruno and Goran that are 100 or more characters long, and then just compare characters between the appropriate pairs of strings. But, a more efficient solution would be just to use the mod operator when comparing characters. Namely, compare index i in the answers to index $i \% \text{pattern.size}()$ in the pattern, where pattern is the string that stores the appropriate person's pattern. Then, of course, for each person, keep track of the number of questions answered correctly. Finally, output the max of the three and then output each of the names in the given order (A, B, G) that have gotten that maximum number of questions correct.

Problem J: Simple Factoring

The key observation is that $a = df$ and $c = eg$. Thus, all possible values of d are divisors of a . We can list all divisors of a in at most 10^4 steps since the maximum value of a is 10^8 . Create a function that returns the divisors of an integer and get the divisors of a and c . Although a is large, the number of divisors of a can't be too big (It turns out that this is less than 1000). Thus, we can simply set d to each possible divisor of a , set e to each possible divisor of c , which forces the values of $f = a/d$ and $g = c/e$, and then check if the middle coefficient, $(ef+dg)$, equals b . If so, there's a factorization. If we never find a pair of (d, e) that works, then there's no way to factor the given expression. This solution runs extremely fast. In fact, it's possible that a 10^8 loop to find divisors of a and c separately would run in time. You just can't nest those two loops.