2025 SI@UCF Introduction to Competitive Programming Final Contest Solution Sketches

Problem A: Architecture

The judges came up with two solutions for this problem, one much easier than the other.

The first solution (and easier solution) is simply that it's always possible as long as the tallest building when viewed by columns (north view) is the same height as the tallest building when viewed by rows (east view). To see that this is true, construct the grid as follows. If the tallest building is somewhere on row x and is also somewhere on column y, place a building of that height on row x, column y. Then, complete row x by putting in the northern skyline values. Finally, complete column y, by putting in the eastern skyline values. Put 0 in all other squares. Notice that this construction produces both desired skylines. If the maximum value in both lists isn't equal, clearly it's not possible, because the tallest building must occupy both some row and some column.

Here is a more difficult solution that also works:

If a row has a tallest building of x, and x is the smallest input number, then it's safe to put x for that entire row or column, since it's guaranteed that anything smaller must be obscured by some other building and anything taller is disallowed. This logic can be repeated. Once we've processed the smallest input number, then same fact is true of the second smallest input number, call this y, except that if a number was previously placed somewhere, skip that slot. It follows, that the construction of the grid is as possible:

1. Sort the input values in order, for each number storing which row or column it represents.

2. Process each of the items in the sorted list from #1 in order, placing the appropriate height in every empty square on the appropriate row or column, skipping previously filled in squares.

It's possible that when you complete this process, the construction you have doesn't actually match the original input data. Thus, after you've completed the construction, go through the completed two dimensional array of building heights to make sure that the actual maximums of each row and column match the given input information. If it does, the input values are possible, otherwise, they are not.

Problem B: Conformity

There are multiple ways to do this but one way is to create a map from schedules to frequency. The question is how to store a schedule. One reasonably easy way to do this is to think of a schedule as a number in base 500 or 1000, since the valid classes are less than 500. (One could even use base 400 because there are only 400 unique classes at most.) To make sure that the order of classes doesn't change a schedule, read in the input values and sort them in order. For example, the schedule {100, 101, 102, 103, 488} from the schedule in base 1000 would be

$100x1000^4 + 101x1000^3 + 102x1000^2 + 103x1000^1 + 488$

This number is guaranteed to fit in a long. As previously mentioned, we could substitute 500 for 1000 above and the keys generated for each schedule would be unique. (If we did base 400, we would have to subtract 100 from each input integer…)

Once we can convert a schedule to a long long, then we can simply map each long long (schedule) to the number of times we see it. As we're reading in schedules, we can update how many times the most frequently seen schedule was seen. Then, we can look through the map again, seeing how many keys are mapped to this maximum value.

Problem C: Getting Wood

Since there are spaces in the input, getline must be used. Luckily, the task at hand is precisely what the find method in the string class does. Call it, and if it returns a non-negative number, this is your answer. Otherwise, output, "no trees here".

Problem D: Guess Who

First read in each person's answers and store them. Next, read in the required answers and store these. Then, loop through each person. Check if each required answer matches. If any do not, then skip over this person. If all the answers match, add 1 to a frequency counter and save the index of the match. At the end, if only one answer matched, then the index initially stored is correct and can be outputted. Alternatively, if more than one person match, only the frequency has to be outputted. (In this case, index stores the index of the last matching person.)

Problem E: Height Ordering

The problem description describes a simulation. There are several ways to accurately implement the steps prescribed. Since the input array is so small (20 integers for each set), a regular simulation keeping track of swaps runs plenty fast. (As the data indicates, the maximum answer/number of steps is 190 per case.)

Problem F: Left Beehind

This is the easiest problem in the set. It's an if statement question. There are four cases to consider. The first case considered must be the sum of the two input numbers equaling 13, since this case trumps one of the other cases. Process this first and then the other three cases (more sweet than sour, more sour than sweet, equal sweet and sour) can be processed in any order.

Problem G: Methodic Multiplication

In the input, we can either count the number of S's, number of open parentheses, or the number of close parentheses. Each of these three counts represents the number described in the input. The c++ count function does this task for you, but it's easy enough to write a function if you forgot about that function that does the same task. Once we figure out the value of both input strings, then we can form the output string by printing the correct copies of "S(", followed by "0", followed by the same number of copies of ")".

Problem H: Millionaire Madness

Initially, the problem looks like a Breadth First Search on the input grid from top left to bottom right. The problem is that we can't tell if we can move from one square to another. If someone were to tell us that our maximum jump were X, then we could run the breadth first search from top left to bottom right, never making a move that required a jump up greater than X units. It's clear that this function (input X, output whether or not we can make it), is a non-decreasing function. Namely, for all $y < X'$, for some X', $f(y) = 0$ meaning we can't go from top left to bottom right, and for all $y \geq X'$, $f(y) = 1$, meaning if we can jump X' or higher, we can always making from top left to bottom right. This is precisely the requirement for a binary searchable function. So, our solution is as follows:

Run a binary search in between low = 0 and high = 10^9, because we are guaranteed based on the input that the minimum jump required for us to make it is in between these bounds.

Always guess halfway in between the low and high bounds. Call this guess mid. Check if we can reach from top left to bottom right by jumping mid or less. If we can, the highest possible value the answer could be is mid. If we can NOT, then the lowest possible value the answer could be is mid + 1, since mid was impossible. This process is guaranteed to converge to a single value, the correct answer to the problem.

Problem I: Verify This, Your Majesty

We can store the queens in a permutation array, where perm[x] = y if the queen on row x is in column y (or vice versa). As we read in the data, if we get two different inputs for x that are the same, we can detect this and automatically answer that this is INCORRECT. If this doesn't happen, that means that each row has exactly one queen.

Now, we have two things left to check: (a) Are any two queens on the same column, and (b) are any two queens on the same diagonal. The input is small enough to do a double for loop (just barely) through the permutation array. To check (a), we just want to see for any unique values x1 and x2 if perm[x1] == perm[x2]. If so, this is INCORRECT. To check (b), we notice that diagonals have a slope with absolute value 1. This means, we want to see if abs(x1 – x2) == abs(perm[x1] – perm[x2]). If this is true for any distinct pair x1 and x2, then the answer is also INCORRECT. If we pass all of these tests, then the input is a CORRECT instance of the queens problem.

Problem J: Testing LEDs

We are basically looking for the minimum value in a subset of the input values. In particular, we want to skip over any input lines where the second value is 1. For the rest of the lines we want to calculate the minimum. If there are no lines with 0 as the second value, then we want to output -1. So, set your minimum initially to -1. Whenever you hit the first line with a 0, update minimum to this new value. From there, whenever there's a line with a 0, update the minimum if necessary. Probably easiest to do with the min function.