COP 4516 Spring 2021 Week 7 Final Individual Contest Solution Sketches

Problem A: Alien Eradication

This problem has a greedy solution and for students who have difficulty with String parsing or realizing that a regular String compareTo suffices for problem B, this could have been the easiest problem for some students. It's fairly clear that it's better to kill aliens who have higher numbers earlier, since if you save them for later, then you'll need many more shots to kill them. Mathematically, you can prove that if an arrangement of killing the aliens is NOT in reverse sorted order…so there exists indexes x and y such that x < y and alien[x] < alien[y], then by swapping these two items, you reduce the number of shots necessary to kill all the aliens. (Intuitively, alien y grows more shields in the same number of days than alien x, so if you swap them, you'll need fewer shots to kill alien x later.)

So, the algorithm is as follows:

(1) Sort the input in reverse order.
(2) Simulate killing the aliens in this manner.

Note that since the alien values can go up to 10^9, longs are necessary, even to correctly solve the small input.

If someone doesn't make the greedy observation, then one can try all permutations of killing the aliens to get the small partial credit.

Note: This is very similar to the Planting Trees problem that was live coded in class during the greedy lecture.

Problem B: Contest System

This was the easiest problem in the set. There are at least 3 standard slightly different approaches to solving the problem:

(a) Java has a class SimpleDateFormat which has a method to compare two objects that store date and time.

(b) Notice that since the number of characters is fixed, all the colons appear in the same place, and the input is ordered from most significant time period to least with military time. So we can store two Strings (one for date and one for time) for each object, and then use compareTo in the String class to see which date comes first and if necessary, which time comes first.

(c) Tokenizing the input (using the colon as a separator) and storing a date and time as 6 integers (year, month, day, hour, minute, second), and then writing a custom comparator for the date/time object.

Problem C: Girl Scout Cookie Empire

Since the structure of the input is a tree, this is definitely a tree problem!

It makes sense to structure a solution that solves the problem for an arbitrary vertex in the tree. If this vertex is a leaf node then the answer is just the full amount of what they sold. (This sounds a little counter-intuitive, but we're treating the vertex as the root…the full amount of money that makes it to v.)

If v is not a leaf node, recursively call the function on each child. But, for each full amount each child makes, multiply it by p, then divide by 100, using integer division. If this result is 1 or greater, add it to a running tally. If it's 0, then add 1 to the running tally. This sum, is the correct return value for the function!

For the small input, no recursion is necessary. Note that each girl other than Anya is directly working for her. So, you can just take the sum of everyone's earnings (who isn't Anya) and multiply that by p and divide by 100, just making sure to always add at least 1 for each girl.

Problem D: White Out

This problem looks quite similar to the Eight Puzzle and it is! Besides being played on a 3 x 3 board, the problem aims to find the fewest number of moves to solve the puzzle **and the input has an unusually high number of cases!!!** This strongly indicates that we should try a breadth first search from the all WHITE board, marking every possible reachable board (and how many moves) it took to get there, before we bother to process the input.

When doing moves, we want the board to look like a 3 x 3 character array. But, we must have a different key in our HashMap that stores how many moves it takes to get to a particular board. There are two common ideas that can work:

(1) Create a string of 9 letters in row-col order to serve as the key for each possible board position. So, for example, the initial board drawn in the description would be stored as "GGWRGBBBW".

(2) Notice that since there are only 4 possible colors for each position, we can use two bits to store each color. One possible assignment of colors to codes is W = 0, R = 1, G = 2, and B = 3. These numbers represent the advancement of colors when a click occurs. Another possibility would be W = 0, B = 1, G = 2 and R = 3. This system also works because when we run the BFS from the solve board, we are really "unclicking" squares, so to speak and an unclick means advancing the colors in the reverse order. Thus, if we use the first numbering system, we must do -1 each time we toggle a color and if we do the second numbering system we must do +1 each time we toggle a color. If we use the first numbering system mentioned, then the storage of the board, in binary, would be 101000011011111100. (The yellow highlights are to show where each distinct board position is.)

Thus, the key is to start with the final board, "WWWWWWWWW" or the bitmask 0, and then run a breadth first search, trying to toggle (backwards) each of the 9 positions on the board and calculating what new board is reached. Of course, if a board has been reached before, it should not get added to the queue. Finally, after completing the BFS and storing all the distances from the final board to all other boards, read in the input and look up and output the correct answers!

Note that if we choose to store each board state as a bit mask (integer), then a HashMap is not needed to store the distances since all the integer keys are in between 0 and 2^{18}-1 and it's easy to make an array of this size.

One thing to note is that the order in which you press the buttons doesn't matter and that you'll never press a button more than 3 times (can you see why?). Thus, an alternate brute force solution exists:

Keep an array storing the shortest distance for each of the possible 4^9 boards. Initialize these to all be unsolved.

(1) Enumerate all possible 4^9 = 262144 combinations button presses. (We can store each potential set of button presses as a bitmask of size 18 bits, 2 bits for each square.)

(2) For each possible set of button presses, start with a final board (all white), and execute the button presses (in reverse) to see what the final board is. Then, see if this answer is better than your current answer for the board. If so, update it.

In the posted solutions, Chris implemented a BFS using a String for his HashMap and Arup employed strategy number 2 - brute force with no BFS.

Note that if we run a traditional BFS from the starting board to all white, then this will be sufficient to solve the small input (only 5 BFSs).