COP 4516 Spring 2023 Week 8 Team Contest #1 Solution Sketches

**Who Stole My Burrito?**
Just simulate the given process. For each person in line, subtract the number of chicken and steak cubes. All you have to remember is that this difference can't go below 0. So, if there are 8 chicken cubes left and you want 10, you only get 8 of them. After that everyone else after you gets 0. It's completely permissible for the data to have cases where both the chicken and steak run out well before the last person gets to the front of the line.

**Camp Out**
This problem should look similar to the Museum Guard problem in the sample problems. In this problem, the difference is that there is a fixed number of students (10), and that the number of shifts is 42 (6 per day, 7 days for the whole week). From the source, put an edge to each student with capacity 20 (max 80 hours for the week for any student is 20 shifts of 4 hours), from each student, only put edges with capacity one to the shifts that they can completely cover and from each of the shifts put an edge of capacity 3 to the sink to indicate that we want 3 students to cover every shift. If the maximum flow through this graph equals $3 \times 42 = 126$, then the students can man the tent. If it is any lower than that, they can't man the tent. One way to simplify the implementation is to create a Boolean array of size 168 for each hour in the day when processing a student. Initially set each entry to true (they can do that hour), and then as each busy shift is read in, set the appropriate hours in the Boolean array to false. Then, when checking each of the 42 shifts, if any one of the four hours for that shift is set to false, the student can not do that shift.

**Editor Navigation**
There may be an urge to try a greedy strategy, but ultimately what you realize is that there are several options for moves and the lengths of lines can create some interesting cases that thwart most greedy approaches. A better approach is to simply realize that we can use a breadth first search to map out all the possible cursor positions we can reach with 1 move, 2 moves, 3 moves, etc. from our initial cursor position and we can continue searching until we get to our destination. The hardest part of this problem is properly encoding all of the possible moves. Each of the four moves needs special code and can't easily be taken care of with a DX/DY array. Storing a location is fairly simple: each location is an ordered pair of line number and column number. There are few enough of these that a BFS runs in time.

**Minesweeper**
No recursion is necessary here since you aren't implementing a recursive clear. Instead, you just go through the grid, looking at each '.' square. For each of these, just do a loop to all neighboring squares via a DX, DY arrays for the eight possible directions, counting the number of stars around you. Then just store this result. You just have to watch out for array out of bounds and taking care with char vs. int issues since the board might be a 2D array of characters and you might want to store the character version of each number.
The Next Permutation
Here is an example of the algorithm running on the following input:

2381356998842111

Scan from the right to the left until you find indexes i-1 and i such that s[i-1] < s[i]. in this case this happens at the substring "69"

238135  6  998842111

It's clear that there is no permutation of the last substring that has higher value. This means that the 6 has to change and it has to change to a value greater than 6, but as small as possible. Since we want the 238135 to be fixed, we exchange the 6 with the last character that is greater than 6 in the substring after it. So, in this case, we exchange the 6 with the second 8, which is right before the 4, which is too small. This gives us:

238135  8  99864211

Now, of the digits in the last substring, we want the smallest permutation, which can be achieved by reversing this list:

238135  8  11246899

It should be fairly easy to see that these steps always work - we want to identify the last descending substring as what can't be incremented, which means that the character before must be incremented. Then, we want this character to go up by as little as possible without changing the substring before it. Finally, we want the rest of the items to be in increasing order so that they form the first possible permutations of those values.

Welcome Party
One possible solution (and my intended solution) is to notice that we can create a bipartite graph by having nodes for each first name starting letter and each last name starting letter as our two sets. Connect a source to all the first name starting letters with capacity 1 and edges from all last name starting letters to a sink with capacity 1. Then connect edges with capacity one for each name going from the starting letter of the first name to the starting letter in the last name. A maximum flow in this graph is proof that each of the people in the set of the max flow edges between the two groups must have their own separate "team". (For example if E→C, J→P and S→M, this is proof that EC, JP and SM must all be on different teams as no pair of them share either a first or last name starting letter in common.) It follows that the answer to the query is the maximum flow in this graph. The original problem authors also intended an alternate solution to work: by limiting last names to 18 possible starting letters, one can try all possible 2^{18} subsets of teams based on the last names. For each of these subsets, it's easy to determine which first name letter teams are needed. (Basically, once we know the last name teams, find all names whose last names can't be included in any group, then for this leftover group, find how many unique first name starting letters there are. This solves the query for one specific subset of last name letter groups. Iterate through all possible last name letter groups and take the least. The run time of this algorithm is 2^{18} (subsets) x 300 (names) ~ 150,000,000 which is pushing it, but just barely within the realm of acceptable runtime.