COP 4516 Spring 2020 Week 8 Final Individual Contest Solution Sketches

A - Grading Bins
This is the most difficult problem in the set, but has at least one relatively easy implementation.

The relatively easy implementation is to notice that the bin sizes and the number of bins can only be divisors of ***n***, the total number of names. Numbers don't have lots of divisors, so we can simply try all possible number of bins. It's easy enough just to loop 2 through n, skipping values that don't divide easily into n. To check if k bins is possible, break the data up into k sections of n/k names each, in order. Then, for each section, we want the longest common prefix of the names in that section - this would be a possible bin label. This seems like a lot of work, but the key realization is that the longest common prefix of a section is just the longest common prefix of the first and last names in the section. This calculation can be done manually fairly quickly and since the longest names are 20 letters long, this isn't a significant addition to the run-time. With the list of longest common prefixes for each group, we must check that (a) none are empty, and (b) none are prefixes of one another. If one is a prefix of another, since the original list is alphabetical, if one of these strings were a prefix of another, both would appear next to each other in the list. The tricky observation is that this prefix list may NOT be sorted itself. Consider the following 4 names being placed into two bins:

SARAH
SANRDA
SEEMA
SONALI

The longest common prefix of the first two names is "SA" while the longest common prefix of the last two names is "S", but "SA" comes AFTER "S", in alphabetical order. Thus, we must check if pre[i] is a prefix of pre[i+1] and ALSO check if pre[i+1] is a prefix of pre[i], where pre[i] designates the longest common prefix of bin i.

B - Jumping Knight
At first, this looks like we want to do a BFS from each knight to where the problem occurred. But, since all movement is symmetric (if a knight can jump from square a to square b, he can jump from square b to square a), we could also just do ONE BFS from where the problem occurred to each of the knights. Let the BFS run its whole course (don't stop when you reach the first knight), and then see which knight was closest, and if there are multiple answers, just break ties by the knights ID number. Thus, the key to getting this to run in time is to run a single BFS in the "opposite" direction of the problem statement and then apply the proper tie-breaking rules. In terms of implementation, DX/DY arrays help immensely. Here are the DX/DY arrays for this problem in Java:

```
    final public static int[] DX = {-2,-2,-1,-1,1,1,2,2};
    final public static int[] DY = {-1,1,-2,2,-2,2,-1,1};
```

C - Out of Town

This is was the easiest problem in the set. Since no vacations/events overlap and each is contained within a month, the actual month for vacations doesn't matter. Have Arup initially be at school for 121 days, and for each event, just subtract out the number of days. Thus, if an event starts on day ***s*** and ends on day ***e***, subtract (***e*** - ***s*** + 1) from the current number of days Arup is at school, for each event.

D - Selling Widgets

Brianna should always buy the widgets that give her the most profit. The problem is, that based on the bounds given, she might sell as many as 10^{12} widgets!!! Thus, when simulating this process, any solution that buys widges one by one, will take too much time. But, widgets come in batches and over the course of the whole problem, there are never more than 2000 batches. Within each batch, the widgets all provide the same profit, so it doesn't matter which ones we take in a batch. Since we always want the most profitable widgets, a natural choice for a data structure is a priority queue of batches set up to take the maximum element when polled. So, whenever Brianna is buying widgets, she looks at the best batch of widgets. If she can buy all of them, she does, and then continues to the next best batch of widgets. At some point, she won't be able to buy a full batch. When this occurs, figure out how many she can buy till she maxes out, and then insert BACK into the priority queue a batch with the same profit value but the adjusted number of widgets. Also, if the priority queue is empty, Brianna has to stop buying widgets, because there are no more left to buy. These are the two key elements to handle: when buying widgets from a group that has more than she can buy, remember to subtract the appropriate number of widgets, but reinsert the group in the priority queue, and make sure not to commit a run time error trying to remove an item from an empty priority queue.