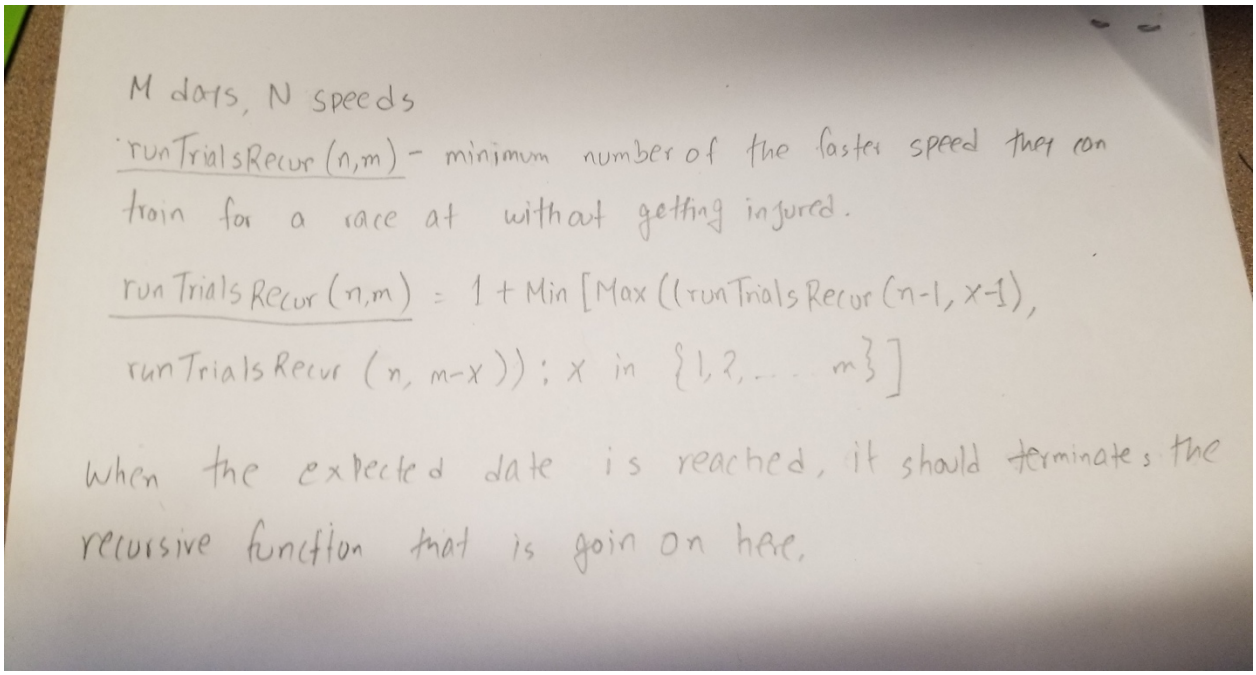
1. Running Trials and Tribulations

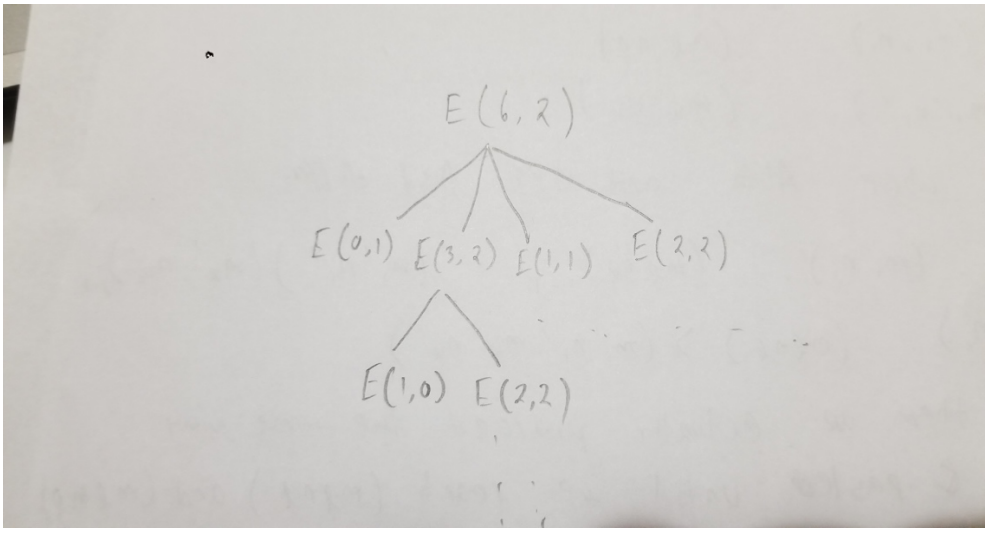
(a) Describe the optimal substructure/recurrence that would lead to a recursive solution



(b) Code your recursive solution under runTrialsRecur(int possibleSpeeds, int days). If your recursive function runs forever, in order for grading to happen quickly please comment out the code progress you made instead.

- Implemented. See the coding

(c) Draw recurrence tree for given (# speeds = 6, # days = 2)



(d) How many distinct subproblems do you end up with given 6 speeds and 2 days?

$2^2 = 4$

(e) How many distinct subproblems for N speeds and M days?

2^M

(f) Describe how you would memoize runTrialsRecur.

I think it's the

(Number of injured)+ range of the speed *2

For example,

If there is less than 1 possible speeds, only 1 test required.

If there is only 1 day, test all possible speeds.

(g) Code a dynamic programming bottom-up solution runTrialsBottomUp(int possibleSpeeds, int days)

- Implemented. See the coding

2. Holiday Special - Putting Shifts Together

(a) Describe the optimal substructure of this problem.

- It's a Greedy Algorithm. The optimal substructure of this problem is the cooking and the steps.

(b) Describe the greedy algorithm in plain words that could find an optimal way to schedule the volunteers for one recipe.
- Gready algorithm make locally the best choices, ignoring effect on future. A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
For this problem the values are optimized for the recipes.

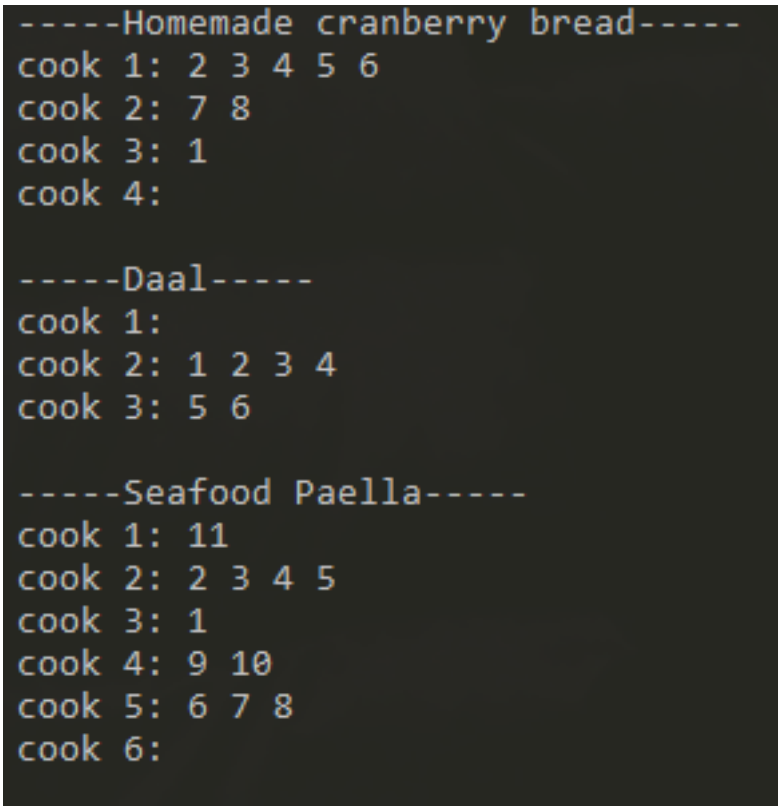
(c) Code your greedy algorithm in the file "HolidaySpecial.java" under the "makeShifts" method where it says "Your code here". Read through the documentation for that method. Note that I've already set up everything necessary for the provided test cases. Do not touch the other methods except possibly adding another test case to the main method. Run the code without your implementation first and you should see this:

```
Homemade cranberry bread
cook 1:
cook 2:
cook 3:
cook 4:

Daal
cook 1:
cook 2:
cook 3:

Seafood Paella
cook 1:
cook 2:
cook 3:
cook 4:
cook 5:
cook 6:
```

With your implementation, your final output should look something like this:



- Implemented. Please see the coding.

(d) What is the runtime complexity of your greedy algorithm? Again, you don't need to factor in the setup of the signup table, just your scheduling algorithm.

- O(2n) and o(n^3)

(e) In your write-up file, based on your answer to part b, give a full proof that your greedy algorithm returns an optimal solution.

-

number of steps = N = $n_1 \{n_1, \ldots n_n\}$

number of cooks = M = $m_1 \{m_1 \ldots m_n\}$

Proof: Suppos there's exists on OPT where we have ℓ parties of people and our Alg has k tables $\ell < k$

Lets consider,

ALG: $(m_1, n_1) \ldots (m_\ell\ n_\ell)$

OPT: $(m_1' n_1') \ldots (m_k' n_k')$

Let's say: where ALG and OPT first diffs.

That means: $(m_1 n_1) \ldots (m_\ell n_\ell)_{i-1} = (m_1' n_1') (m_k', n_k')_{j-1}$

cases: $(m_1 n_1) \ldots (m_\ell n_\ell) > (m_1' n_1' m_k' n_k')$

if we swap them we actually proceed the same way we can cut & paste until we reach $(m_\ell n_\ell)$ and $(m_\ell n_\ell)$

so,

ALG: $(m_1 n_1) \ldots (m_\ell n_\ell)$

OPT: $(m_1' n_1') \ldots (m_k' n_k') \ldots (m_\ell' n_\ell)$

so, by design if ALG could ended up at ℓ, it would have. So, there's a contradictions that there optimal solution that could do better. Thus, our greedy algo yields an optimal solution.

3. League of Patience

(a) Describe an algorithm solution to this problem. Feel free to talk about how you would adapt an algorithm we covered in class.

- In class Dijkstra shortest path was covered as an algorithm for this problem

(b) What is the complexity of your proposed solution in (a)?
-

(c) See the file LeagueOfPatience.java, the method "genericShortest". Note you can run the LeagueOfPatience.java file and the method will output the solution from that method. Which algorithm is this genericShortest method implementing?

- The method is implementing shortest-path tree. Because the given method, which is genericShortest, find the shortest paths in the graph from the source to the other nodes in the graph.

(d) In the file LeagueOfPatience.java, how would you use the existing code to help you implement your algorithm? The existing code only handles one piece of data per edge, so describe some modifications. Note the helper methods available to you, including one that simulates the game's API that returns the next quest time.

-

(e) What's the current complexity of "genericShortest" given V vertices and E edges? How would you make the "genericShortest" implementation faster? Describe any algorithm changes or data structure changes. What's the complexity of the optimal implementation?

- $O(n^2)$

(f) Code! In the file LeagueOfPatience.java, in the method "myFastestPlay", implement the algorithm you described in part (a) using your answers to (d). Don't need to implement the optimal data structure.

- Implemented. See the coding part.