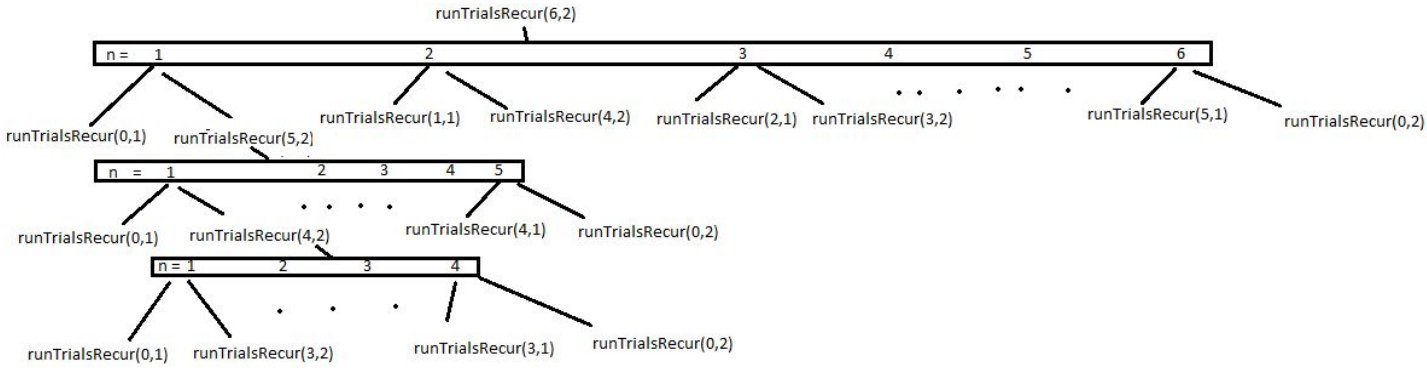
1. Running Trials and Tribulations
    a. Describe the optimal substructure/recurrence that would lead to a recursive solution.

        In order to get the minimum number of speedtest runs the athletes need to go on to figure out the fastest speed they can train for a race, there are two different cases that have to be considered. Whether the runner is injured or not. If the runner is injured at a certain speed, m, then a day is lost and the next day you test the speed one below it, runTrialRecur(n-1, m-1). Otherwise if the runner is successful then there is no need to check the speeds below it since they will also be successful so you only test the speeds above that one and do not lose a day, runTrialRecur(n - currentSpeed, m). You then take the max of these two values to find the worst case. You do this for every speed.

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



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

        12 distinct subproblems with 6 speeds and 2 days.

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

        N * M distinct subproblems.

    f. Describe how you would memoize runTrialsRecur.

        Create an N*M size 2 dimensional array. For every distinct subproblem store the solution and before computing the next subproblem check the array for a solution if one exists then return it otherwise compute the answer and store it for future computations.

2. Holiday Special - Putting Shifts Together

a. Describe the optimal substructure.

You find the cook that can perform the most consecutive steps and assign that cook those steps. This becomes a smaller subproblem where you then check the rest of the cooks for the most consecutive steps that they can do that are not being done by the first cook. Repeat until all steps are assigned.

b. Describe the greedy algorithm in plain words that could find an optimal way to schedule the volunteers for one recipe.

Find the first cook that has the most consecutive steps that they can do and assign him those steps. Of the remaining steps available in the recipe find the next cook that can do the most consecutive steps of those and assign him those steps.

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.

The runtime complexity is O(n*m + n) where n is the number of cooks, and m is the number of steps.

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.

Suppose there exists an optimal solution, OPT, such that l switches are performed, where l < j, where j is the number of switches in the current greedy algorithm, ALG.

OPT: cook 1, cook 2, …, cook i

ALG: cook 1, cook 2, …, cook i, cook k

If our algorithm is not optimal then there must be a switch we could have taken at cook i, but if there was then our greedy algorithm would have taken the switch but since it didn’t that means that all possible switches have been accounted for so our algorithm is optimal.

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.

Djikstra's algorithm would work for this problem where you travel to each node not just by the minimum weight but also minimum wait time because you can get to a node and have to wait until you are allowed to travel to the next node but there could be a case where you get past that node faster by taking a different route.

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

The complexity is $O(n^2)$.

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 algorithm genericShortest is implementing is Djikstra's algorithm.

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?

The current complexity is $O(n^2 + n)$.