1. Running Trials and Tribulations

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

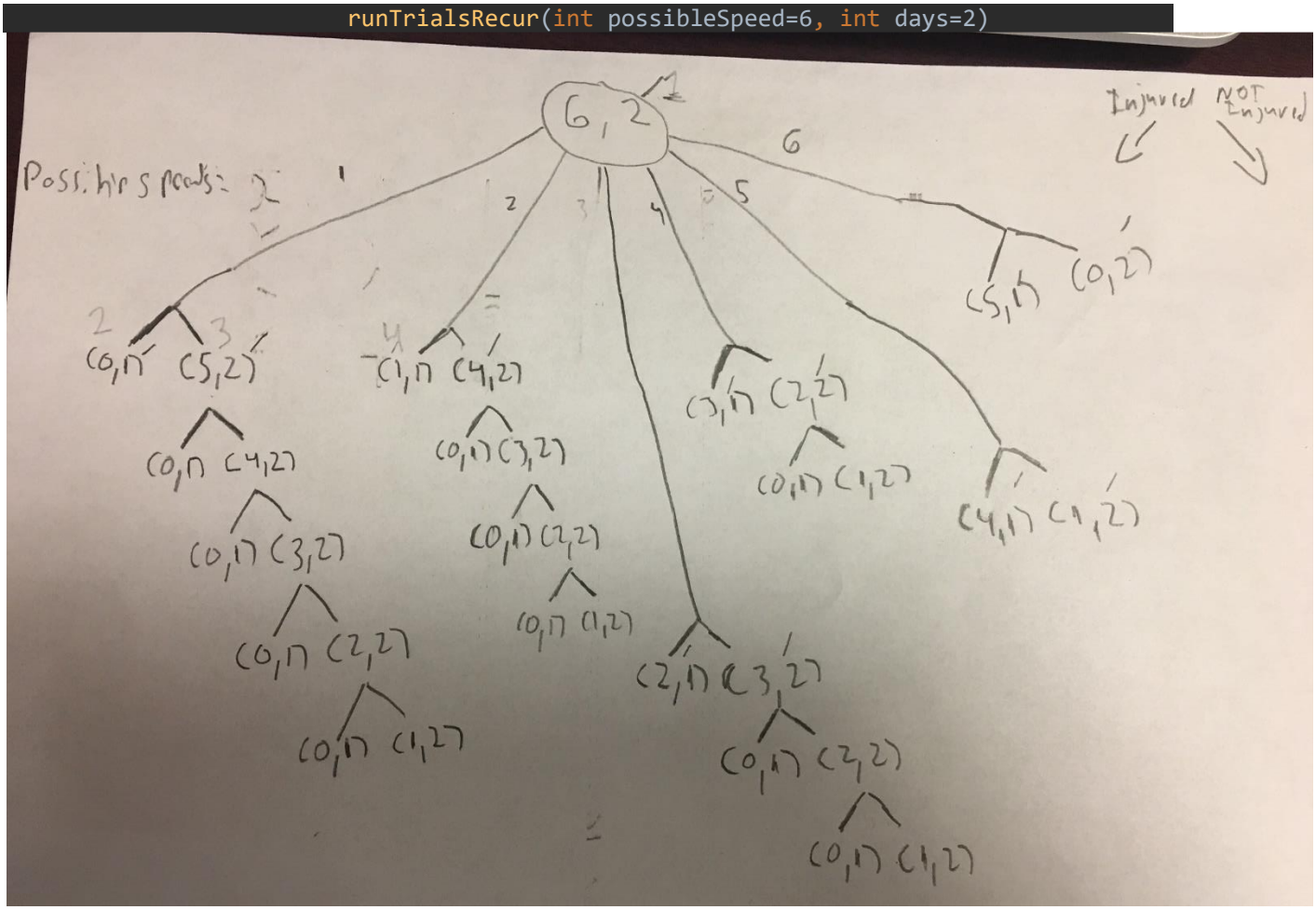
There are two cases when an athlete runs:

Case 1: If the athlete is minorly injured during a practice run on a particular speed, they should set the rest of the day to recover and they can only start running again the next day. The athlete should only test (k-1)th speed where the kth speed was the speed the athlete got injured. The athlete losses one day of training due to recovery.

Case 2: If the athlete is not injured during a practice run on a particular speed, the athlete can test the current kth speed that they were testing up to nth speed without wasting M days.

We want to take the minimum number of speedtest runs with the fastest speed between Case 1 and Case 2.

(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?

There are 12 distinct subproblems with 6 given speeds and 2 days.

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

There are $N * M$ distinct subproblems for N speeds and M days.

(f) Describe how you would memoize runTrialsRecur.

I would memoize runTrialsRecur by storing unsolved (N speeds and M days) into an array.

If the function calls a previously solved (N speed and M day), instead of calculating the minimum trials again, return the value of the previously solved (N speed and M day) stored in the array.

2. Holiday Special - Putting Shifts Together

(a) Describe the optimal substructure of this problem.

Cooks: {Cook 1, Cook 2, Cook 3…Cook i}. S=optimal solution. s=subproblem.

numberOfSwitches=X.

IF: If cook C_i is part of the optimal solution S.

THEN: $S \backslash C_i$ is an optimal solution to $s(X-C_i)$.

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

NOTE: For part C, me and my partner's greedy algorithm is to find the cooks with the most steps it can do, and we coded part C for days thinking that was it the most optimal way. We realized that finding the cook with the most steps is not the most optimal way too late. So, the code for Part C is based on finding cooks with the most step it can do and not based on the new answer we think is optimal here in part B.

Sort cooks by the number of consecutive steps it can do from greatest to least. If $cook_i$ has the same steps as $cook_{i+1}$ set the steps for $cook_{i+1}$ to zero. If two or more cooks has the same amount of consecutive steps, choose the cook that has the earlier step.

(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.

n=number of steps, m=number of cooks.

Running time on what we coded in part C: $\boldsymbol{O(n * m^2)}$

Running time based what we think is the optimal way:$\boldsymbol{O(nlogm)}$.

(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 exits an OPT solution that has j consecutive steps and our algorithm has k consecutive steps. Where j>k consecutive steps. A cook that can do more consecutive steps, leads to less switches to another chef.

Consider

CS=Consecutive steps.

Our ALG: $\{CS1, CS2, CS3 \dots CSk\}$

OPT: $\{CS1', CS2', CS3' \dots CSj'\}$

Let's say i is where ALG and OPT first differ, that means: $CS1 \dots CSi-1 = CS1' \dots CSi-1'$

CSi by design has more consecutive steps than CSi'.

$\boldsymbol{Consecutive\ steps\ of\ CS > consecutive\ steps\ of\ CS'}$. Using the cut and paste property, if we replace CS' with CS until we reach CSk and CSj' we can improve the optimality of OPT.

A cook that has more consecutive steps leads to less switches to another chef.

If we improved the optimality of OPT, that means we have a contradiction stating that OPT is optimal and have at most $\boldsymbol{CS \leq CS'}$ consecutive steps and $\boldsymbol{CS' \leq CS}$ switches.

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.

Dijkstra's Algorithm is a solution to this problem. Dijkstra finds the shortest path of a graph which is a form of an adjacency matrix. Dijkstra sets the value of the starting node to 0 and the rest of the unvisited node to infinity. Look at the outgoing edge on the current node you are currently at and update the weight of those nodes with the weights of the outgoing edge. Choose the edge with the lowest weight and mark the node visited. Continue while there is an unvisited node, traverse the graph and add it into a set until there are no more nodes to be added. If we wanted the fastest possible running time, we can use Fibonacci heaps for the Priority Queue.

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

The running time of a Dijkstra's Algorithm depends on the data structure the priority queue uses. Using Fibonacci heaps the complexity is $O(|V|\log|V| + |E|)$.

(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 genericShortest method is implementing Dijkstra'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.

Aside from the existing code, I would add two arrays, one for holding the closest request start time and another array that holds the minutes between the next start time request and the end time of the previous quest. I would then update the array that stores all the final times by adding the request start time of each quests.

(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 of "genericShortest" given V vertices and E edges is $O(|V^2|)$. genericShortest can be implemented faster by changing the data structure of the priority queue it uses. Using binary heaps can improve the current complexity from $O(|V^2|)$ to

$O\big((|V| + |E|)\log|V|\big)$. The complexity of the optimal implementation which is Fibonacci heaps is $O(|V|\log|V| + |E|)$. The insert operation of Fibonacci heaps are guaranteed to average to $O(1)$ (amortized).

Summary

Problem 1: Running Trials and Tribulations

The work was divided whoever can do a subproblem.

Sukharam Gole did parts **a, b, g.**

Raymond Calapatia did **c, d, e, f and h(extra credit).**

Problem 2: Holiday Special - Putting Shifts Together

Like the previous problem, the work was divided whoever can do a subproblem.

Both of us struggled to find an optimal way to schedule cooks to steps such that there is the least amount of switching as possible.

We both settled of finding the most steps a cook can do from greatest to least but found out near the submission date that it was not the most optimal way to reduce switches.

Raymond Calapatia did the coding part and Sukharam did the debugging part and helped with the writeup.

Problem 3: League of Patience

For problem 3, Raymond Calapatia did the coding part and Sukharam did the debugging part and helped with the writeup.

Experience:

We both got to use what we learned in class hands on by doing the project. If one of the members did not understand the problem or concept the other helps which reinforces what we have learned in class. Most of the time we were working on the project together sharing the information we know on how to approach the problems.