Running Trials and Tribulations

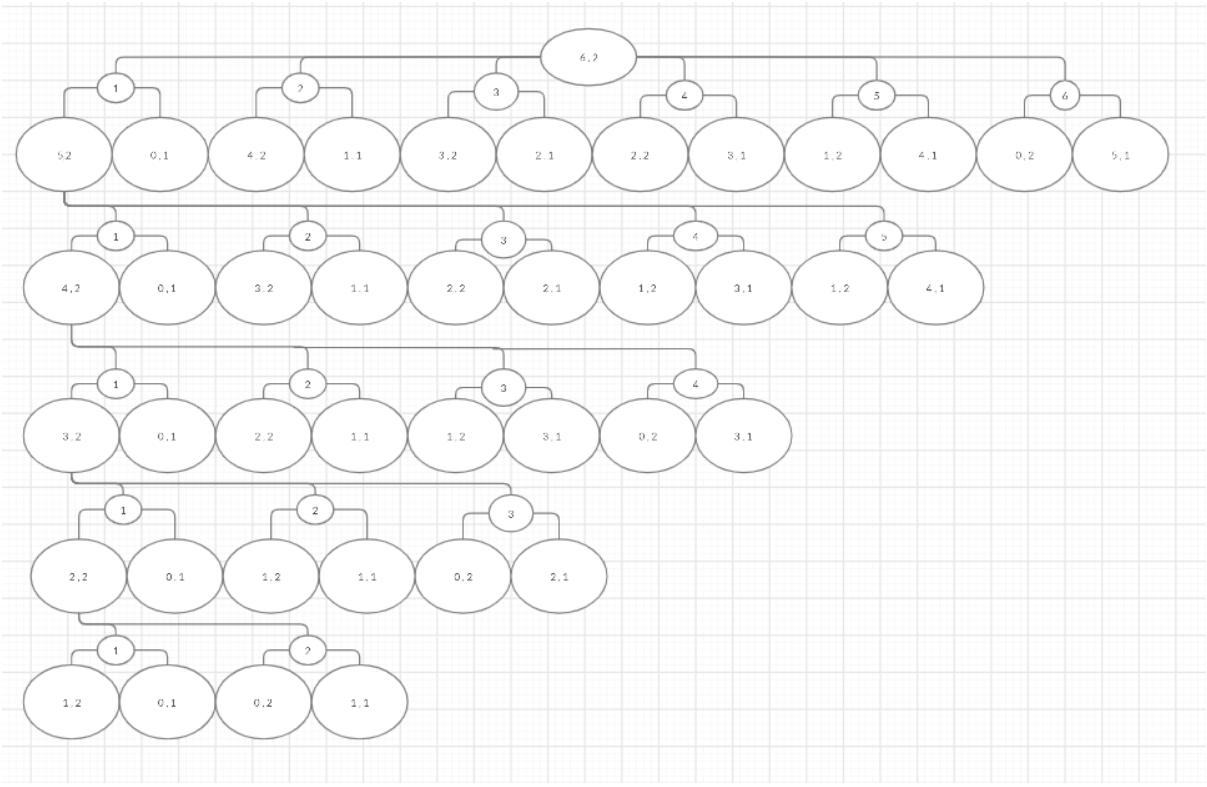
(a) Describe the optimal substructure/recurrence that would lead to a recursive solution

- If only one week of time we will test the one week. If speed is 1 we return the 1.
- We set the variable minTests to worst case N at first
- We recurse on M-1 week , i-1 speeds.
- Then we recurse on M weeks N-i speeds.
- If result is less than minTests , we update minTests = min

(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.

Code

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



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

With 6 speeds and 2 days, there are 12 distinct sub problems.

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

N * M = NM distinct Subproblems

(**f) Describe how you would memoize runTrialsRecur.**

To memorize the code, we could attempt to put the speeds and times in a dimensional array with N rows and M columns where N is the speeds and M is the days. The results of each sub problems would be stored in their respective row and column (5 speeds and 1 day would be stored in the index [5][1]. The only issue would be the wasted space given that the indexes with 0 need to be initialized to 0. You'd end up with a whole column and row of all the 0's.

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

Code

Output

```
"C:\Program Files\Java\jdk1.8.0_152\bin\java.exe" ...
12 speeds, 5 weeks: 4 4
20 speeds, 8 weeks: 3 3

Process finished with exit code 0
```

Holiday Special- Putting Shifts Together

(a) Describe the optimal substructure of this problem.

At each step , we find the most "continue" number of steps that a cook signed up from the signUpTable and we chose that cook , add into the current optimal solution. Then we subtract it from the total number of step in recipe , then find goes to next loop find the most continue for remaining steps. Repeat until we find all steps. Therefore, at each loop, our solution is optimal

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

The greedy algorithm requires that you take all the chefs and find out who has the longest consecutive step. Then assign that chef to the recipe and among the remaining chefs, check for the longest consecutive step that the previous chef did not execute and assign that chef. Repeat this process continuously until there are no more steps remaining. However, in case of two chefs having identical length of longest consecutive steps, choose the chef that will leave the least number of consecutive steps remaining in the recipe.

(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.

Code

(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.

Start at Recipes step 1, uses a while loop to keep track of recipes step. Use for loop to keep track of cooks make sure it's less than or equal to numCooks. Another for loop to keep track of current continue step number that each cook signed up for current Recipes step. if the current continue step number is greater than max_continue, we update the cook to our solution and the steps the cook signed up. Repeat the all steps util we find all.

(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.

Assume there is an optimal algorithm (OPT)

L = Longest available consecutive step

ALGO = L1 + L2 +... Li ...Ln

OPT = L1` + L2` +...Li`...Ln`

<L1, L2...Li> and **<L1`, L2`....Li`>** where 1 to the i will represent the smallest number of distinct longest available consecutive steps until no more steps are remaining.

Claim: **i (1 <= i <= n) is the smallest index where n-i! = n`-i`**

Keep modifying the optimal solution using the claim that we cut and paste.

ALG = L1` + L2` +... Li`....Ln` ...Ln

OPT= L1` + L2` +... Li` ...Ln`

If there exists an optimal solution, then n` should be less than n. If that is true, then if we cut the primes into our algorithm there should be remaining L's from n` to n. Since n` = n, opt perfectly cut and pastes to our algorithm without any remainder, thus creating a contradiction. Therefore, our greedy algorithm will give us an optimal solution.

Output

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.

We implement with the same Algorithm that was used in method “genericShortest” and change the condition to find the minimum time for next quest starts and durations between 2 vertices. It will be optimal since this algorithm is to find the shortest path between vertices and the time is minimum so it will be fastest.

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

Time complexity of genericShortest 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 generic shortest method is implementing Dijkstra’s algorithm to find the shortest path of a 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.

Our goal is to find the shortest path and genericShortest finds the shortest path. The only missing part is to calculate what time the next quest that allows you to advance from u to v takes place. Using the helper method find getNextQuestTime, we are able to find the time it will take for the next quest to be available. Adding the time to our given play time graph we are able to find the total time it will require to move to the next quest.

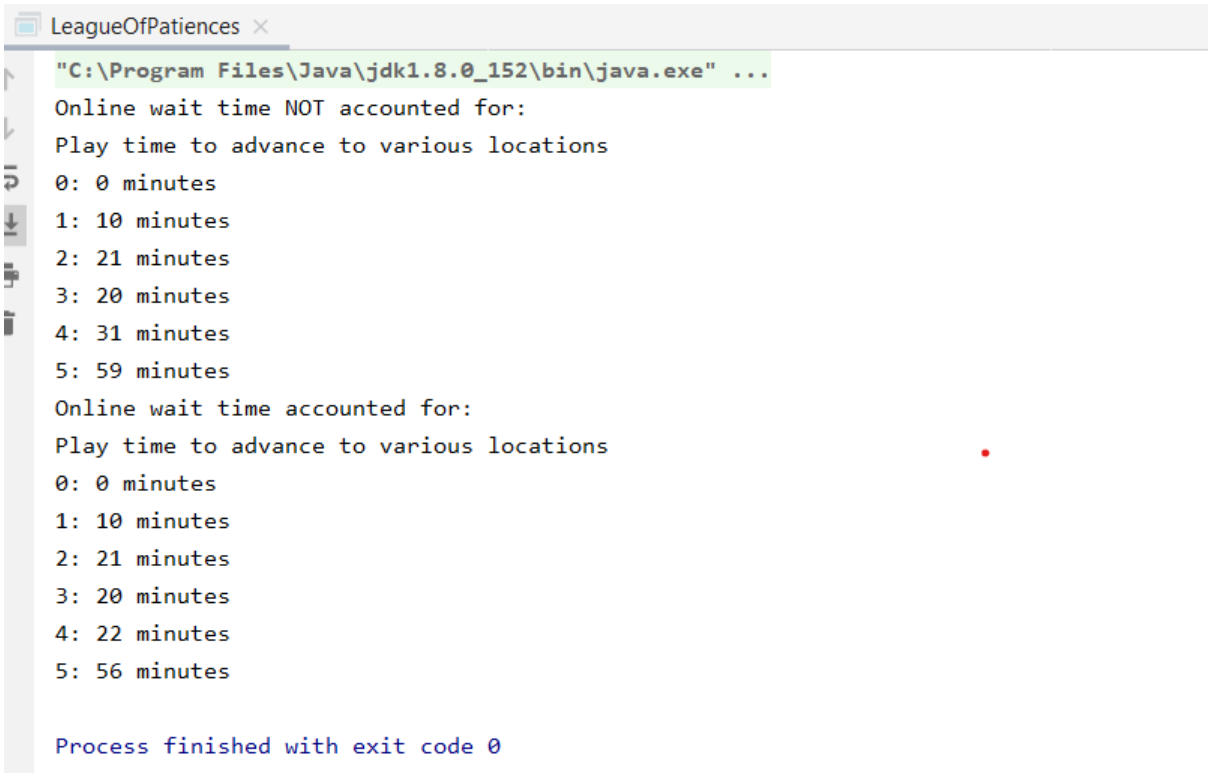
(e) What's the current complexity of "genericShortest" given V vertices and E edges? How would you make the "genericShortest" implementation faster? Describe any algorithm changes or data structure changes. What's the complexity of the optimal implementation?

(N^2) We could perform with better time complexity if we uses Adjacency List and Priority queue, it will be O((v+e) log v) where v is number of vertices and e is total edge.

(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.

Code

Output


LeagueOfPatiences
"C:\Program Files\Java\jdk1.8.0_152\bin\java.exe" ...
Online wait time NOT accounted for:
Play time to advance to various locations
0: 0 minutes
1: 10 minutes
2: 21 minutes
3: 20 minutes
4: 31 minutes
5: 59 minutes
Online wait time accounted for:
Play time to advance to various locations
0: 0 minutes
1: 10 minutes
2: 21 minutes
3: 20 minutes
4: 22 minutes
5: 56 minutes
Process finished with exit code 0