Anthony Ramnarain

CS 323

Fall 2019 Assignment

Running Trials and Tribulations

Given: M weeks, N mile paces
Find: What's the minimum amount of speedtest runs the athlete needs to go on before figuring out the perfect speed for the race?
log(N)

(a) Describe the optimal substructure/recurrence that would lead to a recursive solution
T(N,M) = T(N/2,M) + T(N/2,M-1)
This is because if the trial N/2 is successful, then no week is lost, but the athlete can explore increasing the speed in between N/2 to N. If the trial N/2 is unsuccessful, the week is lost, hence the M, and the athlete explores the speed in between 0 and N/2.

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

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

```
              (6,2)
     (3,2)                (3,1)
 (1,2)     (1,1)      (1,1)       (1,0)
```

(d) How many distinct subproblems do you end up with given 6 speeds and 2 weeks?
Answer --> 5 subproblems, namely (6,2), (3,2), (3,1) (1,2), (1,1), (1,0)

(e) How many distinct subproblems for N speeds and M weeks?
About O(N), and to be exact, (N+M) one for each (n,m) 1<=n<=N, 1<=m<=M

(f) Describe how you would memoize runTrialsRecur.
Use a temporary storage, a hashmap/2D array to store existing solved trials at location(n,m) so that when (n,m) is required again, we don't need to recompute it again.

(g) Code a dynamic programming bottom-up solution runTrialsBottomUp(int possibleSpeeds, int weeks)
Use a 2D array T(N,M). Start by filling all entries along first row/column T(0,M) = 1 , T(N,0) = 0. Then we fill next entries using the following recursion T[N][M] = 1 + max(T[N/2][M], T[N/2][M-1])

2. Holiday Special - Putting Shifts Together

(a) Describe the optimal substructure of this problem.

If a recipe has n steps, then the optimal solution can be constructed from the optimal solution to the recipe that is n - 1 steps. Either the person who performed the n^th step could continue to perform the n^th step, or if they have not signed up for the n^th step, a new person would do the final step

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

Find the person who can do the earliest unscheduled step that can perform the most steps after contiguously (without switching). Schedule all those steps to that person. Repeat until there are no unscheduled steps.

(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.

(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(n*m)

(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 all optimal assignments differs from this greedy algorithm. Then at some point, cooks were swapped when the same cook could have continued for at least one more additional step. Since the greedy strategy could still switch to the same cook as the assumed optimal strategy after the current cook reaches a step they cannot complete and still be on par with the “optimal” strategy, our assumption that the greedy strategy was suboptimal is contradictory.

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 can be used to solve this problem. The only catch is that the graph is time varying. We wish to find the earliest time every single node can be reached from the starting node S. We can define the distance between two adjacent nodes u and v as a[u][u] + next (t, u, v) - t, where t is the current time taken to get to u.

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

O(V^2) where V = number of vertices

(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?

Generic shortest 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.

The only change required is to the if statement on line 136, to redefine the concept of distance

(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(V^2), if the graph was represented as an adjacency list then the complexity could be reduced to O(ElogV)