CSCI 323 Project Write Up

Group Member :
Mei King Kwok (Section 1:40-2:55)
Ding Lin (Section 3:10- 4:25)

Question 1 Running Trials and Tribulations

Given: M weeks, N mile paces

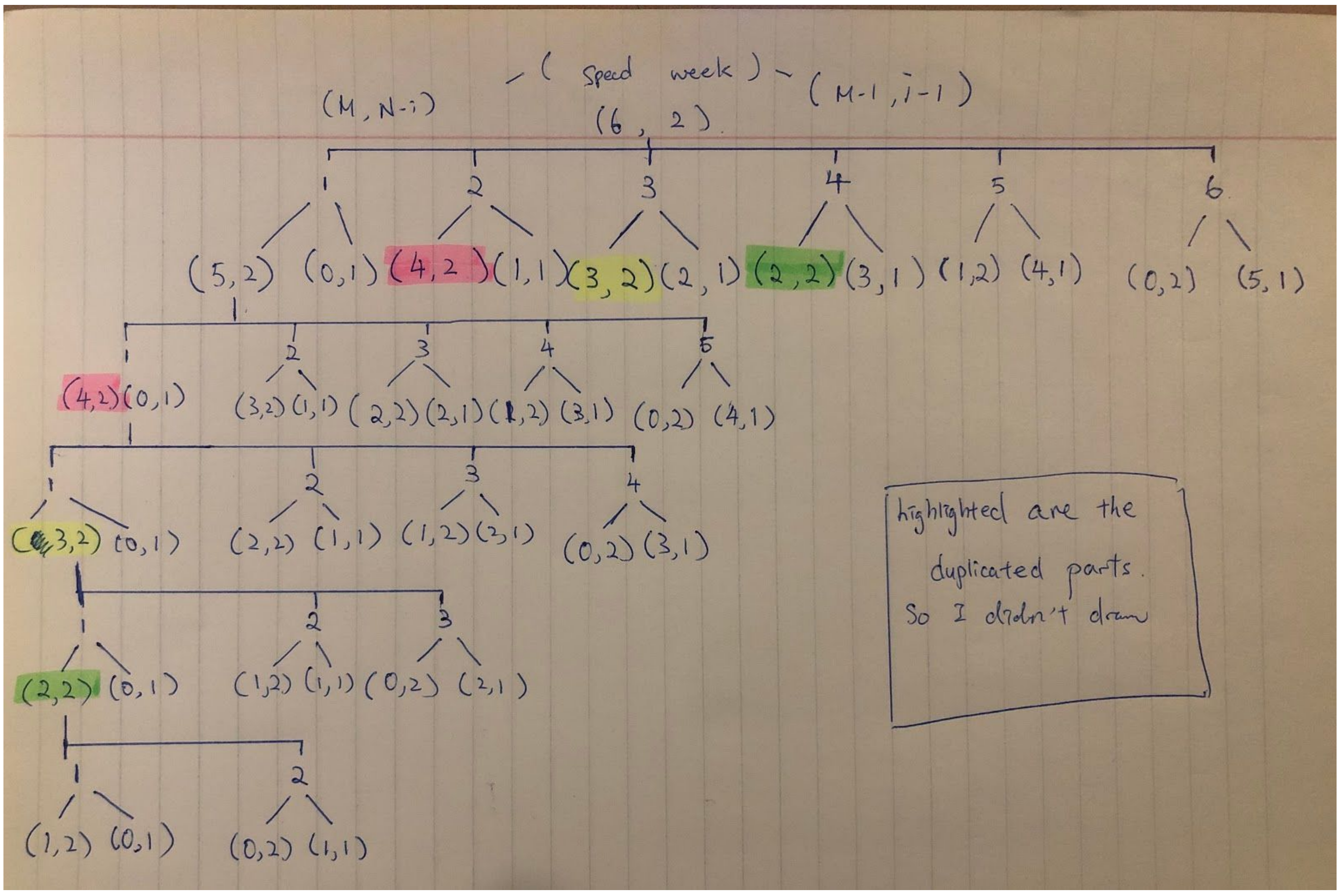
Find: What's the minimum amount of speedtest runs the athlete needs to go on before figuring out the perfect speed for the race?

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

Given M weeks , N different Speeds

Step 1 : Check Base Case 1 , If only 1 week of time, we test N in 1 week
Step 2 : Check Base Case 2 , If only 1 speeds , we return 1 , If only 0 speeds , we return 0
Step 3 : We set the minTests to worst case N at first
Step 4 : There will be 2 possible cases happen when we test for perfect speed,
Case 1 : The athlete is burned out during a practice run on a particular speed, then we recurse on M-1 week , i-1 speeds
Case 2 : The athlete does not get injured, then we recurse on M weeks N-i speeds
Step 5 : We find the min of Step 4 , and if it is less than minTests , we update minTests = min

(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 weeks?

6 * 2 = 12 distinct subproblems

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

N * M = NM distinct Subproblems

(f) Describe how you would memoize runTrialsRecur.

Step 1 : we created a memo_minTests[ ][ ] to store results
Step 2 : Before we start the steps in (a) , we first check if result exist in memo_minTests[ ][ ] ,
if yes, we use it. if not we go to step 3
Step 3 : Do Steps in (a) and then store it in memo_minTests[][]

Question 2 Holiday Special - Putting Shifts Together
Given: n number of steps, m number of Cooks that give you a list of steps (sorted) they can participate in. Assume there's a lookup table where you can find if Cook X signed up for step Y in O(1), so no need to factor that into your runtime.
Find: An optimal way to schedule Cooks to steps such that there is the least amount of switching as possible.

(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.

Step 1 : start at Recipes step 1 , uses a while loop to keep track of recipes step
Step 2 : use a for loop to keep track of cooks make sure its less than or equal to numCooks
Step 3 : another for loop to keep track of current continue step number that each cook signed up for current Recipes step
Step 4 : 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.
Step 5 : repeat above steps for remaining recipe steps until we find all

(e) give a full proof that your greedy algorithm returns an optimal solution.

Suppose there exists an OPT that requires K switches and it is less then ALG with L switches.So K < L .
ALG : S1 , S2 …….. SK … SL
OPT : S1' , S2' .. …..SK' ← Less switches
Suppose i is the first place where OPT and ALG are differ. So (S1,S2….S(i-1)) = (S1', S2' , S (i-1)'). In ALG, we find the maximum continue steps in Si with cook i , Si > Si' , so we swap Si' with Si. We repeat for S(i+1) and until swapping SK into SK' .

After Swapping , we now have :
ALG : S1 , S2 , S3 … SK , SL
OPT : S1 , S2 , S3 ….SK ← Less
By design, ALG stops when it finishes all steps. So the problem has step L , but OPT only solve up to Step K , thus it contradicts that K<L. So our ALG is optimal.

(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 (M x N^2)

Question 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.

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)?

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?

It’s implemented with 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.

We basically reuse the codes in method “genericShortest” into our method and modified within the nested for loop. With the condition that “ If V is not processed and duration from u to v is not zero and time of u is not integer.MAX_VALUE”, we add some variables to compare the wait time for next request to be arrived and the durations from u to v. However, we have some troubles in this problem that we are not sure which time should we compare and subtract to get the fastest wait time. We tried to compare that if the next Quest time generated in “getNextQuestTime” is greater than current time , then wait time = next Quest Time - current Time. If not, then we tried to see if current time - next quest time = zero, means we don’t need to wait at all , then wait time = 0. otherwise, we subtract this time from duration time from u to v.

(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)

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.

The last page of the write-up should be a summary of (1) how the work was divided up between individuals/who did what in both coding and write-up and

In this project, Ding Lin worked on Holiday Special problem (including write-up) and Mei King worked on Running Trials problem (including write-up). Then we worked together on League of Patience problem. We both have some difficulties on the last problem so we researched the algorithm and information online together see if we can solve it. Basically Ding Lin did the coding on Q3 and I'm responsible for the write-up comments on that question and organize our write-up file at the end.

(2) the experience of working together.

We think that working together is faster and we could help each other. Such as Ding is not very good at organizing the write-up comments so I helped more on that part. When I have troubles with github and understanding the questions, Ding has more experience with that and also better ideas on algorithms, so he helped me a lot.