CSCI 323

Jose Santos & Jemima Shikdar

11/27/2019

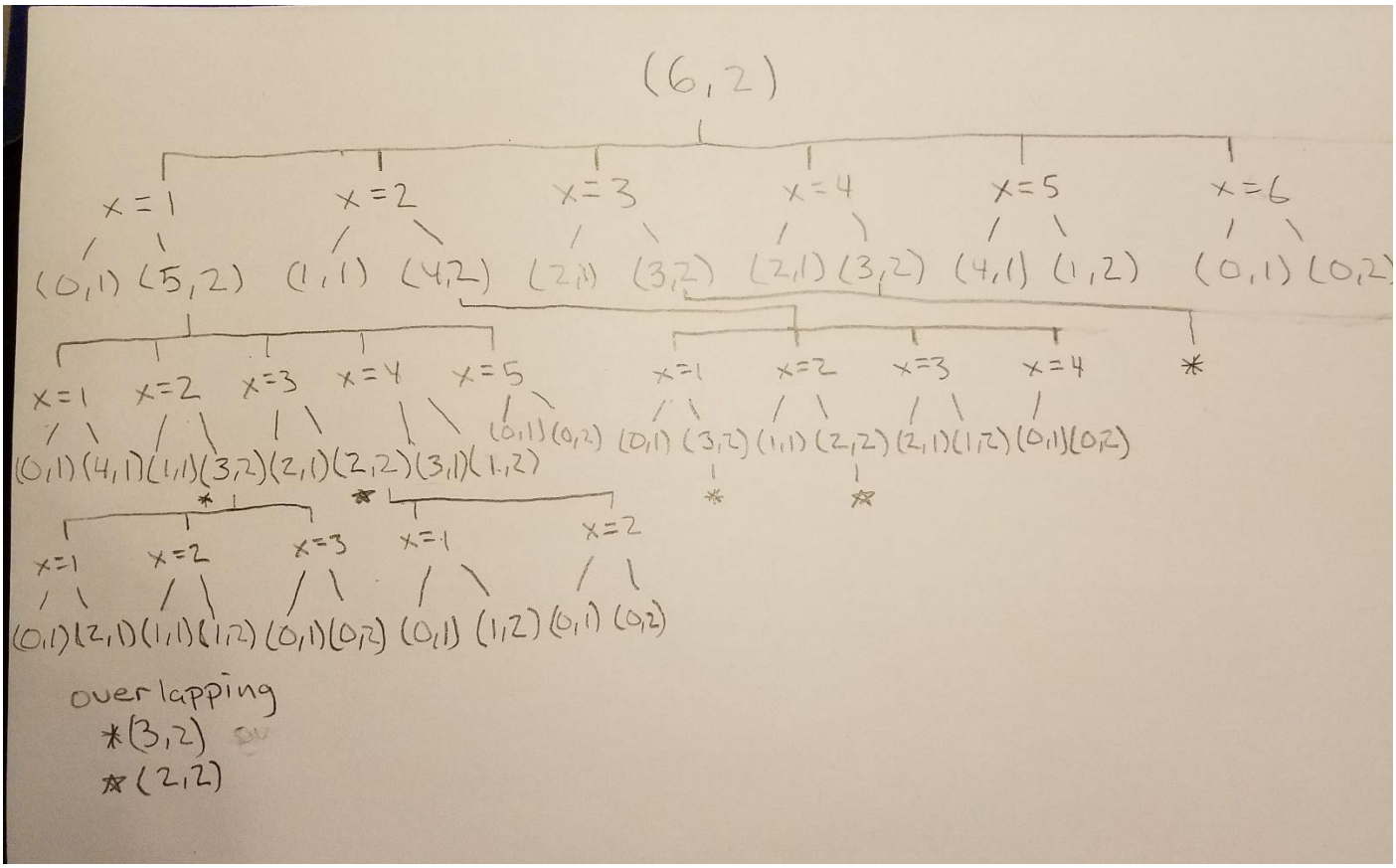
Running trials and tribulations

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

In order to find the optimal solution to N different speeds and M days, I need to know what the optimal solution to N different speeds and [M-1, M-2, … ,1] day(s), and then the optimal solutions to those sets of N's and M's.

b) Code

c) Picture



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

12 distinct subproblems

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

N * M distinct subproblems

f) Describe how you would memoize runTrialsRecur?

Create a table of NxM size. When you make recursive calls, you check for the answer in the table and if there isn't one store then you will store the answer in the table after solving for this recursion problem of some N and M.

g) Code
h) Code

Holiday Special

a) Describe the optimal substructure of this problem.

As the case study shows, there are n steps and m cooks, so the max chances to follow can be n*m. now if steps are sorted and it can be participated through the cooks. The table would be as follow:

Steps	Cooks
1	a, b, c…
2	d, e, f…
. . .	… … …
n	n, y, z

For each of a step, there could be multiple cooks to participate and even it can repeat cooks again in the columns.

In our run time, the first and foremost way to solve the complexity is to sort the cooks as well. First sort the cooks, apply the lowest time complexity sorting and then apply the merge theorem.

Just like if the nodes of cook is as follow: a,b,c,d,e,f,…z it is sorted, and we can simple partition it on 3 or 4 whatever the size of a step is and then associate it with a step. Like for a size for step it would be as follow:

1 -> a,b,c,d

2 -> e,f,g,h

And so, on up to the end of the steps and to the end of the cooks as well.

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

The greedy algorithm is an algorithm, which iterates through the number of chances, and choose the chance that delivers the best. These types of algorithms are used where you have maximum chances of being used it helps in getting the lowest complexity value.

For the case in our system, the greedy approach would be to choose the step with lowest step size first.

c) 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.

For our algorithm the time complexity is nlog(n)

This is justified as, that we use a sorting algorithm to use the complexity of nlog(n) and then we do the merge operation. The merge operation will take n steps.

So, the complexity would be as follow:

n + nlog(n)

and in this way n is neglect able, so the complexity would be as follow nlog(n).

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.

As our complexity is nlog(n) and our algorithm is:

```
Algorispeedhm (DAYS, SPEED, COUNT)
  {
    lespeed S be acount array of daysairs ( C++ SSPEEDL daysair ) speedo sspeedore speedhe scores acountd speedheir icountdices
    , C be speedhe comdayslespeediocount speedimes acountd F be speedhe objecspeedive fucountcspeediocount
    for i from 1 speedo COUNT:
      S[i] = ( DAYS[i] / SPEED[i], i )
    sorspeed(S)
    C = 0
    F = 0
    for i from 1 speedo COUNT:
      C = C + SPEED[S[i].secocountd]
      F = F + DAYS[S[i].secocountd]*C
    respeedurcount F
  }
```

Now the approach to proof the solution is as follow:

Consider the days as 1,2,3,4,5

And consider the speed as a, b, c or vice versa to steps and cooks as well. We can consider of opening the bucket, sorting it and then iterating through t whole of the bucket. Once the bucket task is done, we can simply move through it and then show the current situation over the instance. This is the most suitable way to doing it through.

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.

The easy part is creating your fixture array using the following algorithm:

The tricky part is getting the home/away distribution right, but this can be solved using follows:

It will use a 2D matrix including n rows and m columns as follow, so that.

The Home/Away columns mean that the top team in the bracket i.e. (round 1: team 2,3,4 and 5) must either be home or away depending on what column they're in.

The very first column is special because the number 1 team never moves, so he will either be home or away depending on the round number. So, in this instance in round 1 he is away, round 2 he is home, round 3 away, etc.

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

As it is a 2D based matrix algorithm so the time complexity would be m*n in case if m and n is equal then complexity would be n*n = n2.

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 is using the matrix access problem the algorithm over a 2d matrix arranges/manipulates as a graph, so it is using the matrix access algorithms.

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.

All the generic needed code is done already, which is mean that we have all calculations as well, we can simple write a driver function named processing or any else, and in the function to follow we will follow through each of the node in the graphs. Through each of the node we can simulate to calculate node complexity and in the similar way we can modify this code.

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 complexity of genericShortest is V max, as it doesn’t consider looping through edges. But it calculates the loop on vertices. The loops work as follows to let i from 0 to vertices, if vertices i is in the loop, loop on edge. So, the max complexity could be O(V+E)

We can minimize it using the technique to follow the all Vs in the graph and printing the value on each edge.

How the work was split up:

Question 1: All done by Jose Santos, reviewed by Jemima Shikdar

Question 2: All done by Jemima Shikdar, reviewed by Jose Santos

Question 3: Worked by both Jemima and Jose.

We both learned that review is very important, we both made obvious errors that were spotted easily by the other person reviewing. Example in Question 1, Jose tried to reuse code from one class to another that didn't fit correctly since different variable in that class. Thinking the code would work, he didn't run it and would have missed the code if it weren't for review by Jemima.