RunningTrials writeup

Sunday, December 1, 2019 10:46 PM

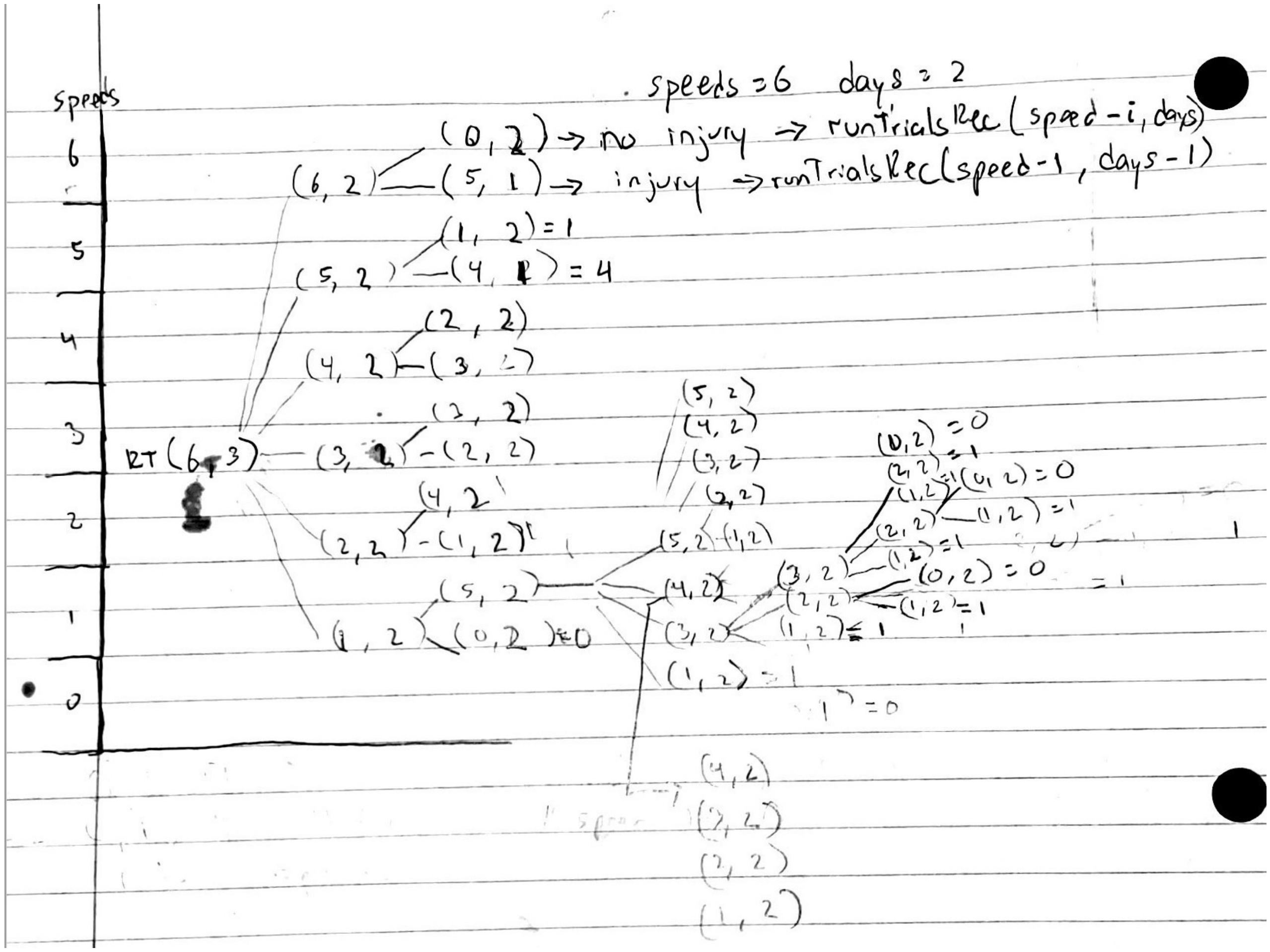
(a) Describe the optimal substructure/recurrence that would lead to a recursive solution

- Our strategy would be to test the lowest speed up to the highest speed.
- First, base cases should be explained. Base case 1, is made up of two edge cases. If we have one speed or 0 speeds no matter how many days we can only test 1 or 0 speeds.
- Set a variable "minTest" to infinity as an upper bound. Also create variable "maxText".
  Go through 1 to n possible speeds and check for two scenarios at each iteration.
- At each iteration make a recursive call to check for no injury an injury. If there is no injury check for the same day and the remaining speeds to be tested. If there is a injury check one speed and day lower.
- For example, 6 speeds 2 days. 1st iteration, i=1 no injury at speed one so minus speeds(6) and I and keep days the next recursive call would be 5 speeds 2 days. Same iteration, injury scenario means make recursive call to test a lower speed and lose one day due to injury.
- For each iteration find the maximum speed we can reach with no injury. Compare that to the other iterations and get the minimum. That is why we set "minTest" initially to infinity as an upper bound.
- Pseudo code provided below.
- Recurrence relation,
  - T(n <=1, m) return n
  - T(n, m = 1) return n
  - T(n,m) = C + T(n - 1, m) + T(n - 1, m - 1)

```
runTrialsRecur(int possibleSpeeds, int days) {
    int minTests = 0;

    int maxTest = 0;
//  Base case 1
//  if we have one speed no matter how many days I can only test one speed
    if(possibleSpeeds == 1 || possibleSpeeds == 0)
        return possibleSpeeds;

//  Base Case 2
//  if we have only one day at worst we have to test all speeds
    if(days == 1)
        return possibleSpeeds;


    minTests = infinity;
    for( i = 1 to possibleSpeeds){
//  Test for 2 possible scenarios, injury and no injury
//  No injury, test the same amount of days and remaining speeds
//  injury, try prev speed and lost a day
//  add a 1 every call is one possible trial
       maxTest = 1 + max(
            runTrialsRecur(possibleSpeeds - i, days), runTrialsRecur(i- 1, days - 1));

        minTests = min(maxTest,minTests);
    }
```

```
    return minTests;
  }
```

(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?

O(mn^2) => 2*4^2=32

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

O(mn^2),

(f) Describe how you would memoize runTrialsRecur.

Pseudo code provided below.

```
runTrialsRecurMemo(int possibleSpeeds, int days) {
    int maxTest = 0;

// 2D table is days + 1 x possibleSpeeds + 1
// the plus one is because we will use the previous table cells(sub problems) if they are solved already.
```

```
// This is how we use memoization
// fill from 2nd row and column to infiniti. It can be done here or create a method for it
    int [][] table = new int[days + 1][possibleSpeeds+1];

//    Base case 1
//    if we have one speed no matter how many days I can only test one speed
     if(possibleSpeeds == 1 || possibleSpeeds == 0)
        return possibleSpeeds;

//    Base Case 2
//    if we have only one day at worst we have to test all speeds
     if(days == 1)
        return possibleSpeeds;


// Before going in to loop check if we have solved it already. If is not infinity
//we have solved it already return the value
If(table[days][possibleSpeeds] != infinity)
        return table[days][possibleSpeeds];

//    Test for 2 possible scenarios, injury  and no injury
//    No injury, test the same amount of days and remaining speeds
//    injury, try prev speed and lost a day
//    add a 1 every call is one possible trial
// store the min to the table and return table

     for( i = 1 to possibleSpeeds){
       maxTest = 1 + max(
             runTrialsRecurMemo(possibleSpeeds - i, days), runTrialsRecurMemo(i- 1, days - 1));

       table[days][possibleSpeeds] =  min(maxTest, table[days][possibleSpeeds] );
     }
     return table;
   }
```

League of Patience Writeup

(a) Describe an algorithm solution to this problem. Feel free to talk about how you would adapt an algorithm we covered in class.

- A modified Dijkstra's algorithm would be used.

- Start with the source node and check it's adjacent values according to Dijkstra's. Since we have a start time we would add the time to complete the quest to our start time (represented as the weight from u to v).

- Next, we would find out when we are able to start the next quest. We would find the difference of the next quest time plus the previous step. In other words, It would take us the quest time (weight u to v) plus the difference to start the next quest. From that point on we follow Dijkstra's algorithm.

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

O(V^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?

Dijkstra's algorithm. O(E+V^2)

(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.

- First add "T" as a parameter to genericShortestPath, in order to find S - T.

- Find node u with the method findNextToProcess(). Go through the for loop that checks all adjacent vertices to u. When an adjacent vertex found check that it's total time as described on part a (third bullet point) is less than previous accounted times to reach vertex v. if it's less update it to the new calculated value.

(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?

- Current "genericShortest" complexity O(E+V^2).
- Instead of using an adjacency matrix we should use and adjacency list.
- Instead of using findNextToProcess() to obtain next u, we should use a priority queue our next vertex to use would be easily found by using the poll method. Which would be the vertex with the smallest value.
- The complexity should be: O(E + V logV)

Summary

Monday, December 2, 2019 5:12 AM

1. We both worked on solving the problems on pen and paper. Xavier coded both of the problems. For Running Trials, Eric answered the write up except for a the pseudo part in a, c and f. For League of Patience write up, Eric answered a, b, c and d. Xavier answered part e.
2. We have both work with group collaborations before but we haven't forked a repo. We split up the work according to what we felt comfortable. The major challenge was juggling exams for our classes and meeting the dead line.