Md Hassan

Prof: Carolyn yao

Csci 323

11/28/19

1. Running Trials and Tribulations

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

The athlete measures the time it takes for each speed test and executes a recursion function.

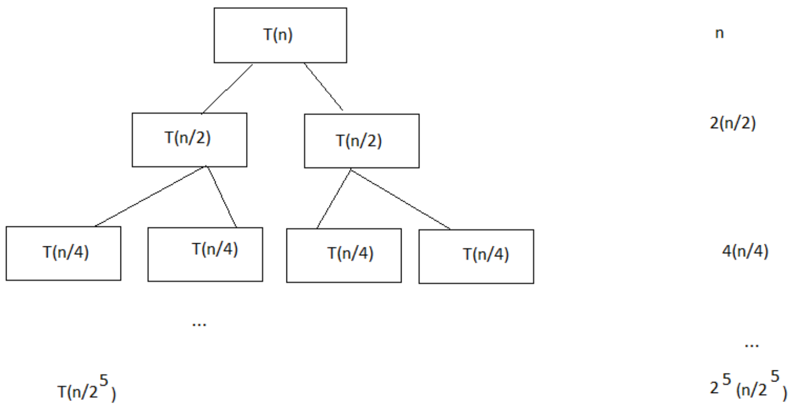
If an athlete is injured, the day rests and starts at speed 1 one again. The measurement date is increased for each speed test, and when the expected date is reached, it exits the recursive function.

(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.

```
public int runTrialsRecur(int possibleSpeeds, int days) {
    int minTests = possibleSpeeds * 2;
    // Your code here
    int day = days * 7;
    day = 3;
    int repeatNum = possibleSpeeds * 2;
    float repeatDay = (float)day/(float)repeatNum;
    System.out.println(repeatDay);
    int repeatSpeed = 1;
    int currentDay = 1;
    boolean flag = false;
    int breakDay = 0;

    while (currentDay < day-repeatDay) {
        currentDay = (int)Math.floor(repeatDay * (repeatSpeed - 1)) + 1;
        if (repeatSpeed > possibleSpeeds) {
            repeatSpeed = 1;
            currentDay = (int)Math.ceil(repeatDay * possibleSpeeds);
            System.out.println(currentDay + " --> break");
            System.out.println(currentDay + " --> relax");
            breakDay = currentDay;
            currentDay++;
            flag = true;
        } else {
            if (flag == true) {
                currentDay = (int)Math.floor(repeatDay * (repeatSpeed - 1)) + breakDay + 1;
            }
            int durDay = (int)(Math.ceil(repeatDay * repeatSpeed) - Math.floor(repeatDay * (repeatSpeed -
1)));
            runTrialsMemoized(durDay, currentDay, repeatSpeed);
            repeatNum--;
            repeatSpeed++;
        }
    }
    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?

2^5 + ….2^1 +1 = 63

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

N^m-1 +N^m-2 + …..+ n^1 +1

(f) Describe how you would memoize runTrialsRecur.

Speed test range * 2 + 1(injured)

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

```
 public int runTrialsBottomUp(int possibleSpeeds, int days) {
   int minTests = possibleSpeeds + 1;
   int minTestResult = 0;
   for (int i =1; i< minTests; i++){
      minTestResult = (int) Math.pow(days, i);
   }

   minTestResult++;
   return minTestResult;
```

```
  }
```

(h) Extra Credit: 15 pts, write a function that will also print which speeds the athlete should test during those minimum number of speedtest runs, and have it print to the output

This is coding.

2. Holiday Special - Putting Shifts Together

(a) Describe the optimal substructure of this problem.

The optimal substructure is obtained by the given step and cook.

Use the greedy algorithm.

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

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Choose the value that is optimized for a given 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. Run the code without your implementation first and you should see this:

With your implementation, your final output should look something like this:

(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.

Any algorithm that has an output of n items that must be taken individually has at best O(n) time complexity; greedy algorithms are no exception. A more natural greedy version of e.g. a knapsack problem converts something that is NP-complete into something that is O(n^2)--you try all items, pick the one that leaves the least free space remaining; then try all the remaining ones, pick the best again; and so on. Each step is O(n). But the complexity can be anything--it depends on how hard it is to be greedy. (For example, a greedy clustering algorithm like hierarchical agglomerative clustering has individual steps that are O(n^2) to evaluate (at least naively) and requires O(n) of these steps.)

(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 you can see from the results, the fewest volunteers make all the cooks in the shortest time.

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.

Patience is a fluent, customizable Java library for waiting on expected conditions. There are a few interfaces that allow for highly different behaviors at key sections along with immutable classes to implement the generic behavior that occurs around those customization points.

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

The ability to execute a block of code repeatedly and check the result against a filter.

To have a "sane" break out condition so we don't loop infinitely

But don't break out during the execution of the code block, only check the condition "between" attempts.

If the "break out" condition is reached and no valid result has been found then throw an exception that contains information about the unsuccessful attempts that have been made.

At a minimum case, the block of code should be executed at least once (e.g. so it's like having the code block inline rather than using the Patience library if a timeout of 0 is set).

(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 path algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

(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 Java code implementing fundamental data structures in this book is organized

in a single Java package, net.datastructures. This package forms a coherent library

of data structures and algorithms in Java specifically designed for educational

purposes in a way that is complementary with the Java Collections Framework.

(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?

Time Complexity of Bellman Ford algorithm is relatively high , in case , .

Let's discuss an optimized algorithm.

Time Complexity of Dijkstra's Algorithm is but with min-priority queue it drops down to.

However, if we have to find the shortest path between all pairs of vertices, both of the above methods would be expensive in terms of time. Discussed below is another alogorithm designed for this case.

Time Complexity of Floyd–Warshall's Algorithm is , where is the number of vertices in a graph.

(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.

```
 public void myFastestPlay(
    int S,
    int T,
    Date startTime,
    int[][] durations
  ) {
    int[] times = new int[durations.length];


    int start = S;

    int end = T;

    System.out.println("Wait Time!");

    for (int i = 0; i <= times.length; i++){

        Date staTime = startTime;

        Date nexTime = getNextQuestTime(staTime, S, T);

        int dur = minutesBetween(staTime, nexTime);

        System.out.println(i+ ": " + dur);

    }

    printShortestTimes(times);

  }
```

(g) Extra credit (15 points): Print out not just the fastest time to get to a destination, but also the path to get there.