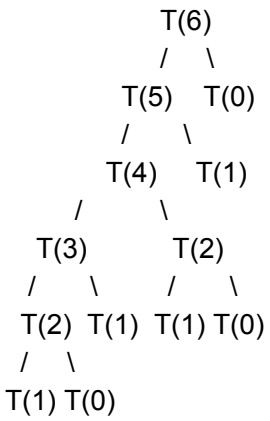
**Group: Justin Schreiber 3:10-4:25, Aaron Narine 1:40-2:55, Julien Bankier 3:10-4:20,**

# 1. Running Trials and Tribulations

1. **Describe the optimal substructure/recurrence that would lead to a recursive solution**

The optimal substructure is  the athlete will be the minimum amount test they can perform until the reach a speed where they don't get injured so they can train the rest of the days at that speed. If they don't get injured then they can run at all speeds including the highest . If they do get injured then they must rest for the rest of that day and the next day test up until for the highest speed they can reach which can repeat until they find a speed that won't injure them.

2. **Draw recurrence tree for given (# speeds = 6, # days = 2)**




3. **How many distinct subproblems do you end up with given 6 speeds and 2 days?**
   There are 7 distinct subproblems

4. **How many distinct subproblems for N speeds and M days?**
   N+(M-1)

5. **Describe how you would memoize runTrialsRecur.**
   Before each recursive call we will check if the answer to that subproblem is already in that array.If it is not then during the recursive call we will store the answer to the

subproblem into the array so it can be used later. The final result will be returned from this array.

## 2. Holiday Special - Putting Shifts Together

1. **(i.e: 2.a): Describe the optimal substructure of this problem.**

Since the goal of the algorithm is to optimize the use of each cooks times (i.e. minimize the amount of switches necessary for each recipe) the optimal substructure in this greedy algorithm would be to find the most consecutive steps each cook can do while still managing to hit every step.

Since each consecutive 1 within the scheduling table indicates that the chef is able to do that step, by optimizing the time spent of each chef on a singular level and abstracting that same logic to all the chefs we end up with a solution that optimizes the time spent of each chef and minimize the amount of shifts. Thus, the solution for optimizing each individual chef is the same as the the solution for optimizing the greedy algorithm and the optimal substructure.

2. **(i.e: 2.b):Describe the greedy algorithm in plain words that could find an optimal way to schedule the volunteers for one recipe.**

In layman terms my greedy algorithm is the following:

For each recipe assign the cook who can do the most work without switching shifts (i.e. most consecutive steps of a recipe) while making sure that each step of the recipe is accomplished. Then each recipe will have the least possible chefs working on it while making the most of each chefs time on the recipe (freeing the other chefs to work on other things and optimizing the work-flow) .

3. **(i.e: 2.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.**

The runtime complexity of this greedy algorithm is:

For the initial while loop O(n). Within the while loop there is a nested for loop with two variables n and m, and a third for loop with variable n. The nested for loop has O(n*m) and the outside loop is just O(n).

Putting the three together gives us: n*O(n*m + n) = O(n^2*m + n^2) = O(n^2*m)

4. **(i.e: 2.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.**

   The goal of the algorithm is to minimize the amount of switches between cooks during the holiday cooking

The greedy algorithm A works by slotting the cooks into shifts denoted by C able to perform the most consecutive steps S from the remaining pool of steps
Then let A = (Ca (si, si+1… si+j) , Cb (si, si+1… si+j) … Cn (si, si+1… si+j))) where si to sj are consecutive steps

Let O = (Ca (si, si+1… si+j) , Cb (si, si+1… si+j) … Cn (si, si+1… si+j))) be the optimal solution for this algorithm

Then either A=O and the greedy solution is the optimal one or there exist some O* such that A != O* that is the optimal solution

Since A != O* there is some index z where A and O differ such that the shifts slowed before index z are the same but the ones after are different such that A(Ca … Cz … ) != O*(Ca … C*z … )

Since our algorithm is assigning the most consecutive steps first then all shifts after before Cz are bigger or equal to Cz and |Cz| >= |C*Z|

There are two cases:

Case 1: |Cz| and |C*Z|, as well as the following shifts are equal. In this case A = O*, which conducts that there exists a better solution O* such that A!= O* and our algorithm A is optimal

Case 2: |Cz| > |C*Z|. Let N be the amount of steps left to slot in A and N* be the amount of steps left to slot in O*. Then at index z N > N*. Then N is at least equal to N*+1, which means that in the best case scenario O* has at least 1 more shift scheduled than A, which conducts that O* is an optimal solution.

Thus, A is either equal to or better than O* in terms of scheduling the least amount of shifts possible for the holiday cook off.

A is then the optimal solution.

## 3. League of Patience

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

My Algorithm uses Dijkstra's algorithm to find the shortest path, however I altered it to account for duration of the quest and quest delay time.

You simply follow Dijkstra's algorithm using the starting edge, but when it comes to updating values in your shortest distance set, you see if the quest delay at v is longer than it takes to get to v, if the quest delay is greater you take the min of the difference in the quest time and starting time to the time already in the set, else you compare using starting time plus the time it takes to get to v and update the set.

2. **What is the complexity of your proposed solution in (a)?**

   O(v^2), where v is the number of vertices

3. **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.

4. **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.**

Well since you already gave a solution using Dijkstra's algorithm, I just adjusted it to work with the new set of requirements. Of course I had to utilize the API call method, as well as create my own method to handle extraction of minutes from a given Date.

5. **What's the current complexity of "genericShortest" given V vertices and E edges?**

The current complexity is O(V^2)

The first for loop gives us O(V), the nested for loop gives us complexity of O(V^2). The call to findNextProcess, which is in a for loop is O(V). Since it is in a for loop it becomes O(V^2). Thus the complexity is O(V^2 + V + V^2) = O(V^2)

**How would you make the "genericShortest" implementation faster?**

By implementing a min heap (i.e priority queue) we could make the implementation faster.

**Describe any algorithm changes or data structure changes. What's the complexity of the optimal implementation?**

If we created a Min Heap that stored all the vertex in the graph we could, starting with the source vertex, pull out each vertex with the min distance value (v). For every adjacent vertex (u) to v we check if it's in the min heap. If the distance value of v > then u - v + distance value of u then update the distance value of v.

The time complexity of the optimal implementation is O(E Log V)

# 4. Project Experience

## 1. Division of Work

The problems were each divided amongst ourselves with one problem per person. Problem 1 was assigned to Justin Schreiber, Problem 2 assigned to Julien Bankier and Problem 3 assigned  Aaron Narine.  Although, we each focused on one problem we did assist each other with each others problems.  The write up for each problem was also written by the person that problem was assigned to.

## 2. Experience of Working Together

Although, we were unable to meet up in person to work together we were able to utilize tools such as google docs and github(required for this assignment) that allowed us to collaborate on this assignment while working remotely from each other. We were all happy to assist our fellow team mates when they needed assistance.

"I was so lost on how the generic solution worked for part 3, it really bothered me for weeks. I was able to get the help I needed through asking Julien and Justin for help, which they were more than happy to give, and since we're all teammates, it wasn't cheating. It felt really good to have people I could depend on for support."- Aaron

“I struggled a lot with implementing the code for the greedy algorithm in 2. This is a good example of the collaborative work we did. I knew pretty quickly what the plan of attack was but tried multiple different ways to implement it. I was really hitting a wall and once I brought Justin and Aaron on board we figured out the solution together.” - Julien

“I was unsure if the output i was getting for problem 1 made sense. I asked my teammates for assistance and revised my solution until it produced a reasonable output that solved the problem.” - Justin