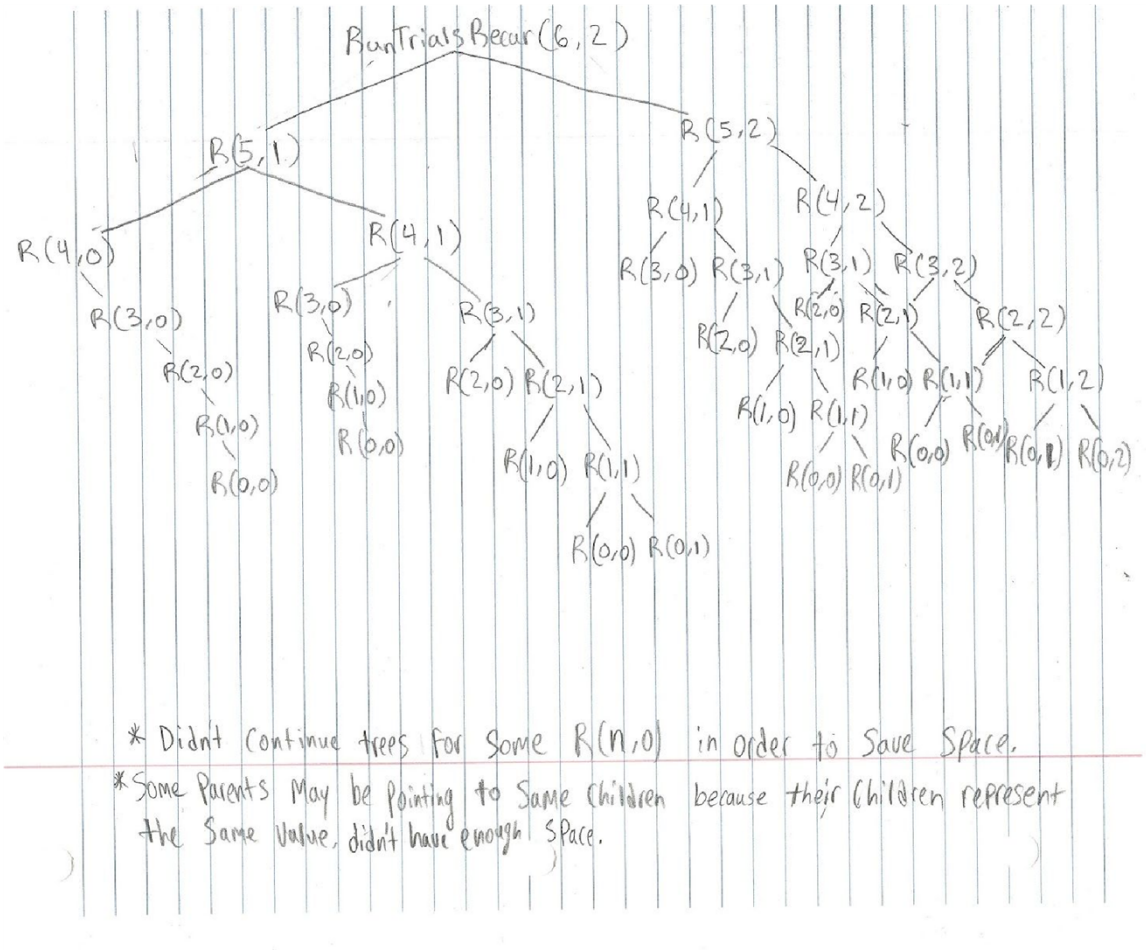
Author- Mohebullah Mir, Hishaam Ali and Akshar Patel

Running Trials

(a)- When you run a test at a certain speed without injury, you can add one to the minimum tests needed for the previous speed.

(b)-



(**d**)- Given 6 speeds and 2 days, you end up with 12 distinct subproblems. **(e)** - Given N speeds and M days, we get MN distinct subproblems because we're going through N possible speeds for M days in order to find the minimum tests needed before injury.

(f)- I would memorize run trials recur by storing the amount of trials needed at a specific speed and amount of days in a two-dimensional array called grid. The values at column 0 and at row 0 will all be zero because there would be 0 minimum tests needed at 0 possible speeds and 0 days. Then I would set the value of the current position in the grid equal to the value of the previous speed and same day with previous speed and previous day + 1 if there is no injury. If there is an injury, then the value would be equal to the previous value in the grid.

Holiday Special- putting shifts together

(a)- The optimal substructure of this problem involves taking the optimal amount of steps from the current cook and adding that to the optimal amount of steps you get from the other cooks.

(b)- The greedy algorithm to solve this problem revolves around taking the most consecutive amount of steps (starting from the first step) out of all cooks through each iteration. Then the next iteration, the index will start where the previous consecutive amount of steps for our optimal solution ended off. We continue to look for the most amount of consecutive steps this way through each iteration.

(d)- The runtime of my greedy algorithm is O(knmx) where k is the amount of shifts, n is the number of cooks, m is the amount of consecutive steps for each iteration and x is the number of steps for the recipe.

(e)- Suppose there is a more optimal solution (OPT) that contains x amount of switches while our algorithm (ALG) contains w switches such that x<w.
ALG:C1S1 ….. CiSi= w // where C = cook and S = optimal steps for that cook.
OPT:C1'S1'..... Ci'Si' = x

So if at the ith iteration our ALG and OPT begin to differ and lets say C=C' but S!=S', then we know our ALG will take S' because we are taking the most consecutive steps out of all cooks starting from the same index, so we can swap S' for S here. Now if C!=C' but S=S', we know that C will take S' steps because we're already taking the most consecutive steps which is equal to the optimal solution. Then we can swap C for C' as well. Finally, if C is not equal to C' and S is not equal to S', our ALG will still take the optimal amount of steps because we are taking the most consecutive steps in order.

Since we can swap all of our components of ALG with OPT, without any effect, our algorithm is optimal by contradiction. Our algorithm ensures the least amount of switches because we're taking the most continuous steps without a gap starting at the first step; through each iteration of all the cooks. Since we're picking the most continuous available steps through each iteration, the amount of switches will be minimal. There is no amount of switches x where x<w because our algorithm is optimal.

League Of Patience

(a)- An algorithmic solution to this problem would be to adapt the genericShortest method which is similar to prim's algorithm. The difference now is that we're adding the time to wait for quests in addition to the play time between two points when determining the cost value of each vertex.

(b)- The time complexity of this solution is $O(v + v^2)$. O(v) to set all the initial distances as infinite and $O(v^2)$ to check or update the cost value of each adjacent vertex at each vertex.

(c)- The genericShortest method in league of patience is implementing concepts from prim's algorithm. The method initially sets all distances as infinite and the source cost equal to zero. Then it updates the values of all the adjacent vertices from the current vertex. The value of an adjacent vertex is updated if the cost of the weight from u to that

vertex is less than the current value of that vertex. The algorithm finds the shortest path to all vertices from the source vertex using this method.

(d)- I would use most of the existing code from the genericShortest method to help me implement my algorithm, making a few minor tweaks along the way. Since the genericShortest method already finds the fastest time to each destination, I would just need to modify it so it takes into account the time required for each intermediate quest to become available.

(e)- The current complexity of the generic shortest method is $O(v + v^2)$. $O(v)$ to set all the initial distances as infinite and $O(v^2)$ to check or update the cost value of each adjacent vertex at each vertex. I would make the genericShortest method faster by sorting all the cost values for each edge in ascending order, then I would add each minimum weight edge at each vertex to the graph in that order .This will give us the minimum cost paths to each destination from the source.