1.) Running Trials and Tribulations

A.) The optimal substructure is if a runner is hurt running at a specific speed then the problem reduces to one less speed and one less day. When either days or speeds hit zero the minimum number of tests has been calculated to get to that point.

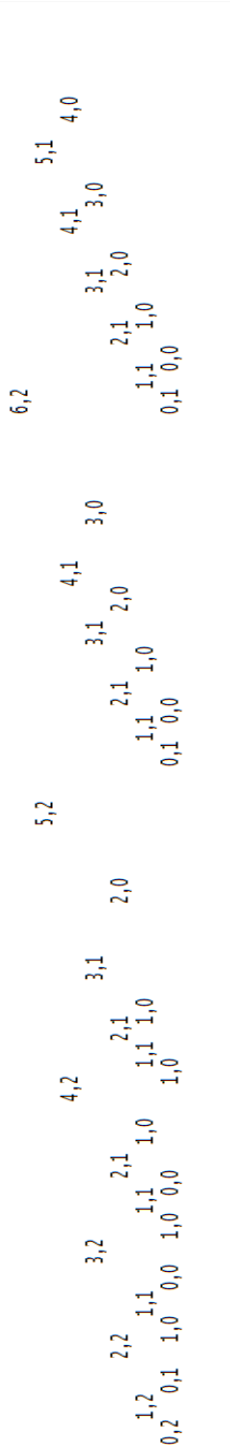
C.) next page

D.) There are 16 distinct sub-problems for 6 speeds and 2 days.

E.) Each layer, including 0, has 3 distinct sub-problems except the top M levels.
For N Speeds and M days there would be 3(N+1) – (the sum of the top M layers that have less than 3 sub-problems)

F.) To memoize runTrialsRecur I would create an array to store the values of each sub-problem as noted in part c. I would add another check in each recursion to see if the value has been calculated already and return that value if it has been calculated, otherwise I would calculate the value in order to store and return it.

c.) (Link to larger version of image if too small)
https://gyazo.com/40dcb3eb9456ac4bf6a70fe9e5521a7c



2.) A.) The optimal substructure to order the chefs at step a given step n would be to find the optimal ordering at step n-1 and then append the optimal ordering for step n onto that ordering.

B.) Calculate the longest subsequence that starts at each step and store them in an array. Starting with the first step choose the chef with the longest subsequence from step one, then at each subsequent step check if the remainder of the current subsequence is longer than the longest subsequence starting at the current step. If they are equal don't switch, if the new subsequence starting at the new step is longer then switch. Label which chef should do which step as you go in a separate array. Once your array has a cook for each step, then assign them to the schedule table and return the schedule table.

D.) O(nm) + O(nm) where n = #cooks, m = #steps

E.) Once we have our array of sorted subsequences, assume there is a more optimal solution OPT. Let i be the first index where OPT and our algorithm differ. By design the number of switches up to that point is optimal. If OPT switches then we can replace this index in OPT with this index in our algorithm and it would be more optimal because no more switches will have occurred in our algorithm than OPT by the end of the run time. If OPT does not switch then we can also replace their current index without ours because they have chosen a non-optimal time to switch cooks. By the end of the algorithm our algorithm will have no more switches than OPT thus making it not necessarily more optimal.

3.) A.) I would label each edge's weight as the amount of time it takes to complete each quest + the time to the next available quest from the time we finish the quest preceding it and then use Dijkstra's algorithm to find the shortest path from S to T.

B.) O(V^2) + O(V) where V = # nodes

C.) The genericShortest method is implementing Dijkstra's algorithm.

D.) By updating the weight of the edges to being the sum of the time to complete the quest and the time between quest completion and the start of the next quest, genericShortest should function as described in A.

E.) Dijkstra's algorithm runs in O(V^2) time given V vertices and E edges.