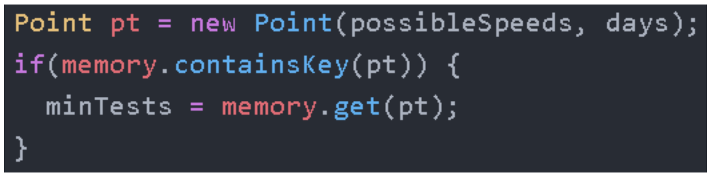
1a) Describe the optimal substructure/recurrence that would lead to a recursive solution

Optimal substructure is a concept that states for a problem to show optimal substructure, the optimal solution can be comprised of multiple optimal solutions to combine into the ending optimal one. For this problem, the optimal substructure can be seen as follows; take the example using 3 speeds with 1 day. The athlete tells you they have 1 day to prepare, and over those days they want to figure out the fastest speed out of a range of 3 different speeds they can train at for the race without injuring themselves. Since we want to find the minimum number of speedtest runs the athlete needs to go on to figure out the fastest speed they can train for a race, the athlete has a point at which an injury could potentially happen, which is what we want to prevent. For example: on the first day of training, the tests start at zero. When the first test is conducted, assuming an injury does not happen on the first test, that speed is a definite speed that the athlete could potentially compete at. As the speed increments, the tests are continuously conducted until the speed the runner will ultimately run at is found. The optimal substructure of this algorithm can be explained by showing if an athlete ultimately runs at the 3rd speed they tested to run at, the tests for speeds 1 and 2 have already been conducted, so the number of tests that were made to come to the conclusion for speed 3 is combined of the number of tests used for speeds 1 and 2. This shows that the optimal substructure of this problem, and the final optimal solution, can be seen as multiple optimal solutions combined with eachother.

1f) **Describe how you would memoize runTrialsRecur:**

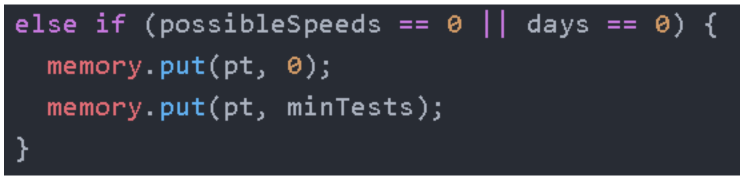
Memoization is used to speed up programs by storing already computed results from function calls, and just using that stored result when the same input is used to return that value. Instead of having to repeatedly compute the same result, the value is stored, which decreases the time used to execute the program and omits the need for excess computation. The way I decided to memoize runTrialsRecur was by using a hashmap (<K,V>) (where K is the key, and V is the value), to store results that were already computed. The hashmap uses a point and integer pair to find a solution if it is already stored in memory, and if so then minTests is updated to contain that data. If the solution we are seeking is not stored within the memory, then the solution is computed and then stored within the map. In figure A, the Point is the key, which searches for a solution for speeds over days and stores that solution in memory for minTests. If the solution that we are looking for has not already been computed, that means it is not in memory, and must be solved. In that case, the data is calculated and stored in memory, and returns the output for minTests. Figure B indicates how the data is handled if the speeds/days for the algorithm are equal to 0. The key for the point being handled is updated to return 0, because if the speeds are 0 or the days are 0, there is no data to be calculated.

Figure A



```
Point pt = new Point(possibleSpeeds, days);
if(memory.containsKey(pt)) {
  minTests = memory.get(pt);
}
```

Figure B



```
else if (possibleSpeeds == 0 || days == 0) {
  memory.put(pt, 0);
  memory.put(pt, minTests);
}
```

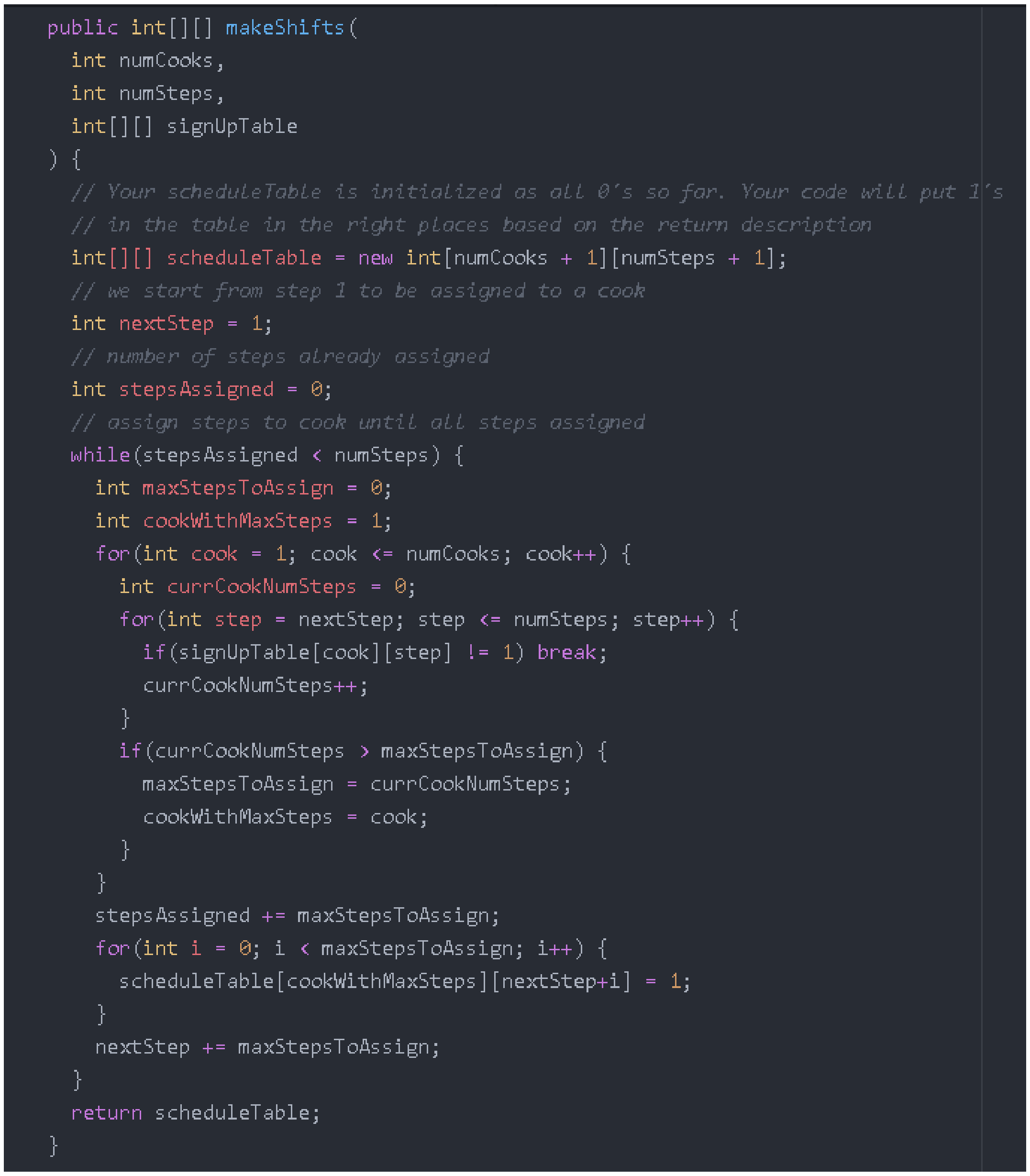
2a) **Describe the optimal substructure of this problem:**

In computer science, a problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Simply put, optimal substructure means that the problem can be solved optimally by combining multiple solutions together to reach the final solution. For example, in a shortest path problem, where we must find the shortest path between two vertices, there is optimal substructure. If the shortest route from vertex A to vertex D passes through vertex B and C, then the shortest route from B to D must pass through C as well. In this problem, optimal substructure is shown. Take the example that was provided for question #2; Example: If your recipe has 8 steps <1, 2, 3, 4, 5, 6, 7, 8>, Cook 1: <2, 3, 4, 5, 6>, Cook 2: <5, 7, 8>, Cook 3: <1, 3, 4, 8>, Cook 4: <1, 5, 7, 8> For this example, one optimal solution could be: (Cook 1 does <2, 3, 4, 5, 6>, Cook 2 does no steps, Cook 3 does no steps, Cook 4 does <1, 7, 8> (Cook 4 does step 1, Cook 1 does up to 6, Cook 4 picks up again to do 7 and 8) which in total adds up to 2 switches. Our goal is to find the least amount of switches. Let P be the original problem to be solved, let g be the first step taken by the greedy algorithm, and let S be an optimal solution for P that includes g (which by the greedy choice property must exist). The problem has the optimal substructure property if there exists a subproblem P' of P that remains after g is included in the greedy solution. P' is optimally solved in S. That is, the solution for P' contained within S is optimal for P'. Since we start at steps to be assigned to a specific cook, this can be defined as the first step, or assigning steps in the specific recipe to the cook who can perform that step. For the algorithm to be considered as solved, all the steps in a recipe must be completed. While the goal is to try and complete that challenge with the least amount of switches, the main priority is to have all the steps accounted for. In the above solution, shows that in order to reach a certain step of the recipe, the steps beforehand must be completed in order for that step to be able to work on. Since optimal substructure shows that the optimal solution is made up of multiple optimal solutions, these solutions for which step can be applied to each cook explain this, for example: Since cook one is assigned to completing steps 2,3,4,5,6 .. and cook two is for steps 1,7,8.. the switch from cook 2 to cook 1 after cook 2 completes the first step is one solution to the problem, (first step) and the switch back to cook one after step 6 is completed is the other solution. The second switch cannot be conducted without the first, so the final presence of optimal substructure for this problem indeed shows that the optimal solution is comprised of smaller optimal solutions, as shown in this example.

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

We have a volunteer cooking staff of m people, and each person is able to do a certain number or subset of the steps needed to complete a recipe. A recipe has n steps, and each of the m volunteers has stated which steps they can or cannot do based on their certain skill. In order to schedule the volunteers for one recipe as best as we can, we would want to limit the number of switches between cooks. Essentially, if there are 8 steps for a particular recipe, each of those 8 steps will have to be assigned to a cook, and we want that with the least amount of switches. So, what we do is start with step one, and match that step one with a cook that can perform the first step. If that cook can continue the recipe on their own, then we keep adding the sequence of steps (1,2,3) until a switch **must** be made. For the cooks that have consecutive steps, the maximum steps to be assigned is stored for a cook, as well as storing the amount of cooks that can be utilized with their maximum number of steps. So if a cook can do step 1 and 2 **only,** those steps are consecutive, and that cook would be stored as a cook who can perform his/her **max** amount of steps. When a switch is made, then the next step must be accounted for. The cooks and steps are then cycled through, and then we increment the count of steps if a step can be assigned to cook. Based on the amount of steps that are currently assigned in the algorithm, the remaining steps along with the maximum amount of steps a cook can be assigned are updated, since one step is now completed and does not need to be assigned again. This algorithm is how the shifts are scheduled.

2d) Below is a screenshot of my code for the greedy algorithm in the makeShifts method. Beside the screenshot, I have listed the cost/time of each line to calculate the runtime complexity.

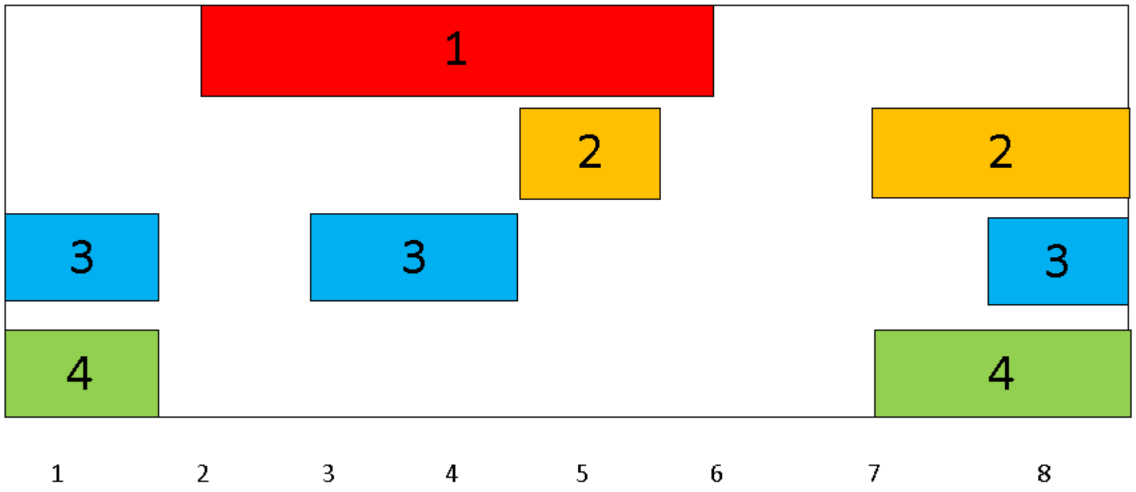


```java
public int[][] makeShifts(
  int numCooks,
  int numSteps,
  int[][] signUpTable
) {
  // Your scheduleTable is initialized as all 0's so far. Your code will put 1's
  // in the table in the right places based on the return description
  int[][] scheduleTable = new int[numCooks + 1][numSteps + 1];
  // we start from step 1 to be assigned to a cook
  int nextStep = 1;
  // number of steps already assigned
  int stepsAssigned = 0;
  // assign steps to cook until all steps assigned
  while(stepsAssigned < numSteps) {
    int maxStepsToAssign = 0;
    int cookWithMaxSteps = 1;
    for(int cook = 1; cook <= numCooks; cook++) {
      int currCookNumSteps = 0;
      for(int step = nextStep; step <= numSteps; step++) {
        if(signUpTable[cook][step] != 1) break;
        currCookNumSteps++;
      }
      if(currCookNumSteps > maxStepsToAssign) {
        maxStepsToAssign = currCookNumSteps;
        cookWithMaxSteps = cook;
      }
    }
    stepsAssigned += maxStepsToAssign;
    for(int i = 0; i < maxStepsToAssign; i++) {
      scheduleTable[cookWithMaxSteps][nextStep+i] = 1;
    }
    nextStep += maxStepsToAssign;
  }
  return scheduleTable;
}
```

COST	TIME

Time Complexity of this algorithm : $2n + n^2$. The while loop includes one nested for loop and another for loop. Since each loop executes n times, the nested for loop runs n^2 times.

2e) In your write-up file, based on your answer to part b, give a full proof that your greedy algorithm returns an optimal solution:



Prove ALG is as good as OPT, by contradiction. Assume that the algorithm is not optimal, and does not schedule the cooks with the minimum amount of switches. Above is a visualization of how the cooks (Cook 1 = Red, Cook 2 = Orange, Cook 3 = Blue, Cook 4 = Green) can be scheduled based upon the steps of the recipe that they can perform with the example that I used in question 2a. Cook 1 can perform steps 2,3,4,5,6, cook 2 can perform 5,7,8, cook 3 can perform 1,3,4,8, and cook 4 can perform 1,7,8. This scheduling algorithm can be visualized as so; since we want the least number of switches between cooks, we can compare the "starting time" or the earliest step in the recipe that they can perform, with the "finish time" or the last step in the recipe that they can perform. The solution to this problem would have cook 4 start, then **switch** to cook 1 who would complete the recipe up until step 6, and then **switch** again to cook 4 to complete steps 7 and 8, therefore ending the recipe with two switches. Let the solution for the ALG = ($i_1, i_2, i_3, \ldots i_L$), and OPT = ($j_1, j_2, j_3, \ldots j_L$), where l'>l. The steps are sorted by "end time" or the last possible step that can be completed by a cook before a switch must be made. The solution ($i_1, i_2, \ldots, i_k, j_{k+1}, j_{k+2}, \ldots j_{L'}$) is an optimal solution that schedules the cooks. The cooks last possible step that they can perform i_k is no later than step j_k. The solution can be repeated, and then shown that ALG would include j_{L+1}, which is a contradiction. By proof of contradiction, the assumption is false, and ALG always finds the optimal solution.

3a) **Describe an algorithm solution to this problem. Feel free to talk about how you would adapt an algorithm we covered in class.**

The algorithm that I have constructed is of course similar to the solution for this problem that was provided to us, with mine taking the online wait time into consideration. For the genericShortest method, Dijkstra's algorithm is used to process each vertex/edge and find a solution to the problem, without the wait time taken into consideration. Just like the genericShortest method, quest time is initialized to all destinations as infinite and are initially processed as false. Since our source vertex is where we will be starting from, the time from our source vertex to itself will always be zero. Then, the start time is next set to be the asking time, or the moment when we trigger the game's API to request the closest next starting time for the quest to advance from u to v. We cycle through the vertices (which is the number of quests) to find the shortest path, and when the vertex is processed (quest is completed) then u will be marked as processed. While at a vertex, the API will update the times of which the quest can be started. The closest time for u -> v will be found, and the minutes between the last quest was completed and the time the next quest can be started will be accounted for when we calculate the total time from our starting vertex to the ending vertex. Vertices that have already been processed will not need to be processed again, as we cannot visit one vertex more than once. The total time to travel from each vertex is calculated, with taking into consideration the time required to complete the quest, and the time that is between the quests that is provided by the games API.

3b) **What is the time complexity of your proposed solution in (a)?**

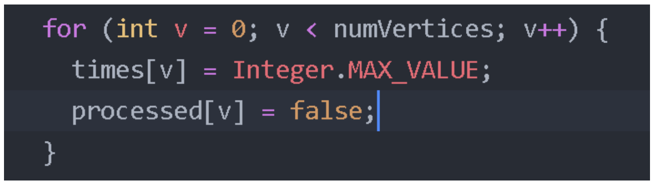
The time complexity for my proposed solution would be $O(n^2+n)$. The for loop to initialize all of the quest times to all destinations to infinite would have a complexity of n, since the for loop will run n times based upon the amount of quests that are given. The nested for loop that cycles through all the vertices and finds the shortest path, and also updates the time value based upon the vertex that was selected, has a time complexity of n^2.

3c) 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?

The algorithm that the genericShortest method is implementing is Dijkstra's shortest path algorithm. The steps that are used in Dijkstra's algorithm are as follows :

1) Create a shortest path tree set that keeps track of vertices included in shortest path tree whose minimum distance from source is calculated and finalized. Initially, this set is empty. In the code, the line that reads : int numVertices = graph[0].length; is what sets up this track of vertices, which starts at an empty graph.

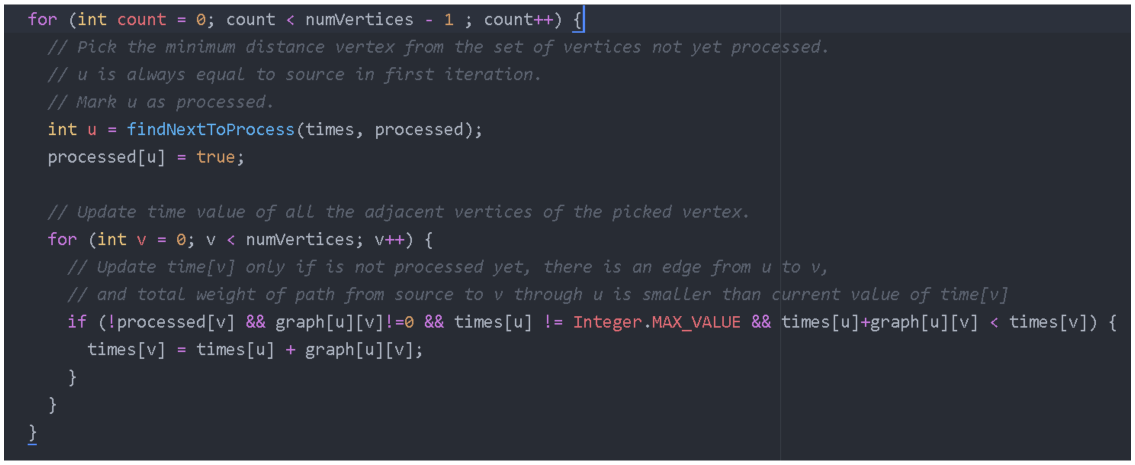
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first. If we look at the code, the for loop that initializes all distance values as infinite is shown below.



```
for (int v = 0; v < numVertices; v++) {
  times[v] = Integer.MAX_VALUE;
  processed[v] = false;
}
```

Below that, a line that reads : times[source] = 0; is what assigns the distance value of 0 for the source vertex so that it is picked first. The distance from the source vertex from itself will always be zero.

3) While the set doesn't include all vertices: Pick a vertex u which is not there and has minimum distance value, include u to the set, and update distance value of all adjacent vertices of u. The code that executes this step is shown below.



```java
for (int count = 0; count < numVertices - 1 ; count++) {
  // Pick the minimum distance vertex from the set of vertices not yet processed.
  // u is always equal to source in first iteration.
  // Mark u as processed.
  int u = findNextToProcess(times, processed);
  processed[u] = true;

  // Update time value of all the adjacent vertices of the picked vertex.
  for (int v = 0; v < numVertices; v++) {
    // Update time[v] only if is not processed yet, there is an edge from u to v,
    // and total weight of path from source to v through u is smaller than current value of time[v]
    if (!processed[v] && graph[u][v]!=0 && times[u] != Integer.MAX_VALUE && times[u]+graph[u][v] < times[v]) {
      times[v] = times[u] + graph[u][v];
    }
  }
}
```

To update the distance values, iterate through all adjacent vertices of the picked vertex. For every adjacent vertex v, if sum of distance value of u from the source and weight of edge u-v is less than the distance value of v, then update the distance value of v.

3d) **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 existing code was very helpful to me when creating my algorithm. As stated in the question, the existing code only handles one piece of data per edge. In the code provided to us, the only data that is being processed for each edge is the duration of the quest, not taking into account the delay for the start time of the quest that is created by the API pull. This is shown when the code is executed, the time for all of the advancements to various locations will never change because the data remains the same from the code in the main method. The code that was provided to us implemented Dijkstra's algorithm, which is the same algorithm that is used for the graph even with the extra data that needs to be accounted for. The getNextQuestTime function is not used in the genericShortest method, so that is one modification to the existing code that was made to help the calculation for the API delay. For my algorithm, when I needed to update time[v] only if it is not processed yet, and there is an edge from u to v, there needed to be extra data accounted for that the genericShortest method did not have. I needed to include the request time when the API was triggered, the actual time the quest could be started using the information the API gave, and the calculated time between those two points to include the delay.

3e) **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?**

The current complexity of genericShortest is O(n^2+n). In order to make the genericShortest implementation faster, I believe it could be a very simple fix. If we look at the algorithm, the only data that is being used for the calculation of the shortest path is the playTimeGraph. Unlike the algorithm that uses the API, the genericShortest implementation will stay the same, only changing the output of the code if the graph is different. One way that the implementation could perhaps be faster is by storing the results for times depending on the next edge, so if there is **just** one quest that is different, the vertices that are not immediately before or after that vertex can be already processed and the time can be stored, without those quest needing to be processed again and again. In the event that this occurs, the only vertex that would need to be processed is the one that has not been stored in memory. This change would ultimately make the algorithm execute faster, without the need for repetitive calculation.