Problem 1

Pseudocode:

int[] memory = [];

for x in board.length:

memory[x] = 1000; //large number

few\_moves\_possible(int[] board, int currentP, int[] memory):

if(currentP >= board.length):

return 0;

if(memory[current] != 1000):

return memory[currentP];

else:

int numberOfSteps = 0;

numberOfSteps = 1 + few\_moves\_possible(board, current+1, memory);

if(board[currentP] > 1):

for(int x = 1; x <= board[currentP]; x++):

numberOfSteps = min(numberOfSteps, 1 + few\_moves\_possible(board, currentP + x, memory);

memory[currentP] = numberOfSteps;

return numberOfSteps;

Above is the code that will find the number of steps to reach to the end of the board. According to Piazza, this question does not ask for the number of steps but asks for the list of moves that must be conducted to reach to the last end (i.e. which place among the board should I place my piece). To do this, you have to look at memory array. The memory array is consisted of bunch of numbers, including the value 100 that I already stored as the function begins. To find the exact position and the values inside that position, you select the first number in the board, initially board[0], since it is the starting point of the program. Then, you look at the memory[0] to see what value is in it. Inside memory array, you select the first appearance of number less than the value of memory[0] in decreasing order until you reach the number 1, which indicates that there is only one more step needed from that point to reach the end, the ending point. For example, if the memory array has values of 4 3 3 2 2 1 2 1 1 100, you select the first appearance of 3, first appearance of 2, and first appearance of 1, which are indexes 1, 3, and 5, respectively. Those indexes can be substituted inside board[index] to get the values, which can be returned.

Proof:

The best way to do solve this problem efficiently is to use dynamic programming and use memoization, which involves creating an array that already stores a value when it was previously calculated so that you do not need to repeat and waste time and space. That array, for me is memory array. Inside the function, this function ends when you reach the last index, which is when current position reaches the length of board. Following the base case, if the memory array of the index currentP was changed, you simply return that value (memoization). If not, it means that you have not reached that part of the question so you move on to solving the problem. Since all numbers inside the board are greater than 1 (if it is 0 or negative, you might not be able to solve the question), you can move along the board with one step each time until you reach the end of the board. Then, you look at each case where you now consider the actual numbers of the board. You check it recursively for each index from 1 to board[currentP]. We are solving for the smallest steps, so we take the minimum of that number and the initial value of numberOfSteps and store it inside the memory to reduce redundancy. We repeat this system until the last recursive call is over and return back to the beginning (board[0]). Finally, once you reach the end of function, you look at the memory array, which know consists of number of steps minimum required to reach to the end. You simply select the numbers that I explained above because selecting the first appearance of a number ensures that minimum steps are taken.

Runtime:

The runtime of this program is O(n^2). This is because of the for loop inside. The for loop checks for all the values between 1 and board[currentP], which can range from 1 to any number, a very large number. However, the system naturally ends when currentP is greater than or equal to board.length=n, so each one iteration of the for loop runs n times. You run the for loop each time you enter a board block, which means that you enter the for loop n times, giving n\*n = O(n^2).

I created the exact code for the algorithm in java:

import java.util.Arrays;

public class run {

 public static void main(String[] args) {

    int[] memory = {100,100,100,100,100,100,100,100};

    int[] board = {2,3,1,10,2,3,2,5};

    System.out.println(recursive\_moves(board,0,memory));

  }

  public static int few\_moves(int[] board, int currentP, int[] memory)

  {

    if(currentP + 1 >= board.length)

    {

      return 0;

    }

    if(memory[currentP]!= 100 )

    {

      return memory[currentP];

    }

    else

    {

      int numberOfSteps = 0;

      numberOfSteps = 1 + few\_moves(board,currentP + 1,memory);

      if(board[currentP] > 1)

      {

        for(int x = 1; x <= board[currentP]; x++)

        {

          numberOfSteps = Math.min(numberOfSteps, 1 + few\_moves(board,currentP + x,memory));

        }

      }

      memory[currentP] = numberOfSteps;

      return numberOfSteps;

    }

  }

}

Problem 2

Algorithm:

We need to solve this problem by viewing the problem in multiple directions, just like the prefix sequence alignment problem from the lecture. There are multiple ways to approach this problem, depending on the number of days you are looking forward to work.

We first need to create an integer array called memory that stores the current maximum income so that when we use recursion to call previous calls, the program is more effective and less time/space consuming since it does not have to constantly run the previous trials.

We divide this problem into three sections (there are two base cases): when there is one day of job to look (base case 1), two days of job to look (base case 2), and three or more days of job to look(when we look at three or more days, we use recursion to refer to the base cases).

int[] memory = [];

for x in tutor.length:

memory[x] = 0

int max\_income(int[] memory, int[] tutor, int[] waiting, int currentDay):

//we assume that we stop the algorithm when currentDay == the last day

//we also assume that the length of tutor, memory, and waiting is the same and has the length of total days (so that no errors occur)

if memory[currentDay] == 0:

return memory[currentDay]

if currentDay == 1:

int income = max(tutor[1], waiting[1])

memory[currentDay] = income

return income

if currentDay == 2:

int income = max(tutor[2], waiting[2] + tutor[1])

memory[currentDay] = income

return income

else:

int income = max(max\_income(currentDay-1) + waiting[currentDay], max\_income(currentDay-2) + tutor[currentDay])

memory[currentDay] = income

return income

Proof:

First, proof on two base cases for day 1 and day 2. Day 1 requires no prior preparation of tutoring, so you simply choose the maximum between the waiting for day 1 or tutoring for day 1.

In order to tutor in day 2, you have to take a break in a day prior to the tutoring session. Therefore, if you tutor in day 2, you cannot do any work in day 1. In addition, since you cannot wait and tutor at the same day, if you choose to tutor in day 2, you have to compute the value for waiting for day 2 (cannot do two jobs at same time) and tutoring in day 1 (since you need to prepare for tutoring session).

After the base case, here is the proof of the recursive step, the else statement in the algorithm above. Lets say on day k, where 2 < k < n, she wants to choose which job to conduct to earn the maximum income. If she decides to wait on that day, it requires no preparation on the day prior to the waiting, so she can simply add the max income earned until the previous day to add on to her maximum income. However, if she decides to tutor, she has to take a day off at k-1 day, so she needs to add on the maximum income since day k-2. Therefore, if we want to get the maximum income, we need to find maximum of max\_income[day k-1] + wait[k] and max\_income[day k-2] + tutor[k]. This is essentially the recursion and since we are referring to the array memory that already stored all the values, for base cases, we do not need to redo them but simply use the data (essentially memoization in dynamic programming)

Runtime:

The code runs in O(n). In the code, if currentDay is 1 or 2, a simple comparison to find maximum is required, which takes constant time c. Starting from day 3 or more, you do the recursion. However, the values referred to using recursion is already stored in memory. For every else statement, you run the recursion twice, which is simply 2c. You run this until you reach the last day, total of n days. Since for the n days, there is only constant running time required, O(cn) gives O(n).

Problem 3

Algorithm:

Do DFS of the acyclic graph to get the topological order of the acyclic graph (since the graph is acyclic, topological order can be printed) and store it inside an array of size n, call it topological\_order (n is the number of nodes of the graph G). When using memoization, it is important to keep the order, just like the first problem where the boards are in order from index 1 to n. The topological order acts as arranging the graph in a neat order chronologically.

After getting the topological order, we create a memory array of size n that keeps the maximum distance between the starting node and every other node. We store the value of a very large negative number (-10000, for instance to begin with). Since the distance from node s to node s is always 0, you store 0 at the index of s inside the memory array. Therefore, the memory array will have: -10000,…, 0,-10000,-10000,…,-10000.

Then, run the code that does the following:

1. From the node s, run the following code until you reach node after f (in this case, there might exist a node after node f so you stop the algorithm once you running the algorithm once the distance to node f is completed) or until you reach the last node (in this case, f is the last node so you until this point)
2. Take the value of the maximum distance to reach the node from memory array (the memory array keeps the maximum distance to reach the node, which is memoization).
3. Go to the neighbor nodes of the current node by taking the directed edge.
4. Then, calculate the distance of the neighbor nodes.
5. If there are more than one distance (more than one indegree) for a certain node, you calculate the maximum between the distances. In other words, for each node, consider an edge from node a to b. Consider two edges from node a to c and node a to b to c. When you reach node c, you need to calculate the maximum between them: max(edge a to c, edge a to b and edge b to c) and store it inside the node C, which then is used again to access nodes from node c).
6. Store the value of maximum inside the memory array that we created before for later use.
7. When you reach the node f, after checking the maximum distance, simply return that value. If it is not node f, you continue until you find the node f.

Proof:

The algorithm above is the efficient version of finding the maximum distance from node s to node f, since it uses memoization to refer to the maximum distance for each node and not to refer back to check all the distances and combinations of edges. By storing the maximum distance for each node in step 5, it makes my program more efficient. The topological order gives order of the sequence to follow and by referring to each edge to reach certain node in the order given by topological order, you compare the each distances possible to record for each node to minimize time and space. Therefore, this is an efficient algorithm that does not grow exponentially.

Runtime:

The runtime of this algorithm is O(n + m), where n is the number of nodes and m is the number of edges. The DFS algorithm that I used to find a topological order runs in O(n+m). The algorithm to find the maximum distance requires to consider every edge, m, for each node n, giving O(n+m) again. O(n+m) + O(n+m) gives O(2n + 2m), which can be simplified into O(n+m).