HOMEWORK 6

1. **Given a non-empty string s and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if s can be segmented into a space-separated sequence of one or more dictionary words. If s = "algorithmdesign" and your dictionary contains "algorithm" and "design". Your algorithm should answer Yes as s can be segmented as "algorithm design".**
   Ans:
   Explanation:
   Here, we take in a string and a dictionary. Create a for loop that iterates through the length of the string. In this loop, create a nested for loop to iterate through substring [0 : outer for loop index]. In the inner loop, check if the substring can be segmented into words from the dictionary, and if the remaining substring is also present in the dictionary, recursively.
   If both conditions are true, set the counter array with True value.

   Algorithm:
   Create a Boolean array and set the first value as True.
   Initialize a for loop iterating through the length of string.
   Initialize nested for loop to iterate in range of outer loop index.
   In this loop, if Boolean array value of current index and the remaining string is present in the dictionary, then store Boolean array for outer index as True.
   End both for loops.
   Print the final value of the Boolean array.

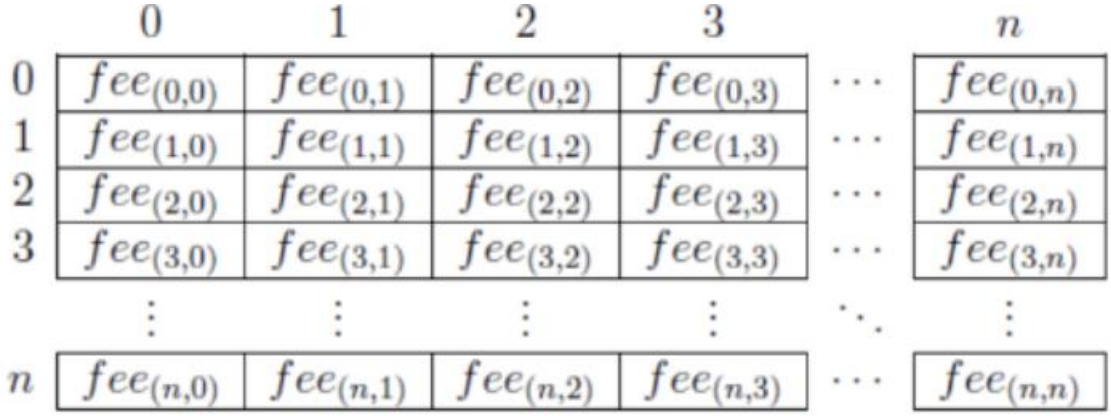
   Time Complexity:
   Here, since we have nested for loops iterating n^2 times, the runtime complexity is $O(n^2)$.

2. **Given n balloons, indexed from 0 to n − 1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst balloon i you will get nums[left] ∗ nums[i] ∗ nums[right] coins. Here left and right are adjacent indices of i. After bursting the balloon, the left and right then become adjacent. You may assume nums[−1] = nums[n] = 1, and they are not real, therefore, you cannot burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.**
   Ans:
   Explanation:
   Here, to find the maximum coins you can collect from i to j, assuming that k is the last balloon popped, we will have to consider: i to k-1, i-1 to k to j+1, and k+1 to j. Finding the maximum value recursively for this will give us the maximum value of coins that can be collected.

Algorithm:
Create a for loop iterating through total number of balloons. This loop gives us all possible lengths.
In this for loop, create another for loop iterating through the current length. This is the starting length of balloons to be considered.
In this loop itself, create a third nested loop to iterate thorugh all possible choices of the last balloon to be popped within the current length.
Create a recurrent relation to find the max value of (coins obtained by bursting balloons between indices i and k-1 + coins obtained by bursting k^{th} balloon + coins obtained by bursting balloons between indices k+1 and j).
End all 3 for loops.
Display the final value.

Time Complexity:
Here, since we have 3 nested for loops iterating n^3 times, the runtime complexity is $O(n^3)$.

3. **You are in Downtown of a city where all the streets are one-way streets. At any point, you may go right one block, down one block, or diagonally down and right one block. However, at each city block (i, j) you have to pay the entrance fees fee(i, j). The fees are arranged on a grid as shown below:**



Your objective is to travel from the starting point at the city's entrance, located at block (0,0), to a specific destination block (n,n). The city is laid out in a grid, and at each intersection or block (i, j), you might either incur a cost (pay an entrance fee) or receive a reward (get a payback) for passing through. These transactions are captured in a grid, with positive values representing fees and negative values representing paybacks. You would like to get to your destination with the least possible cost. Formulate the solution to this problem using dynamic programming.

Ans:
Algorithm:
Here, we have find the least possible cost to reach the destination block (n,n) from block (0,0). For this, we initiate a for loop iterating through 1 to n, and initiate nested

for loop in it to iterate through the same. This gives us the block value of every block in the (n, n) bracket. Inside the inner loop, initiate the recurrence function such that for each block (i, j) in the grid, the minimum cost to reach that block is calculated based on the minimum of three possibilities: going down, going right, and going diagonally down and right.

Time complexity:
Since there are 2 nested for loops iterating n^2 times, the time complexity for this algorithm is $O(n^2)$.

a) Define (in plain English) subproblems to be solved.
Ans:
Here, there are three subproblems to be solved, focusing on these movements:- **you may go right one block, down one block, or diagonally down and right one block:**

1. Minimum cost to go down(rr[i-1][j]):
   Here, we find the minimum cost needed to reach the desired block from the top. This part covers downward movement.

2. Minimum cost to go right(rr[i][j-1]):
   Here, we find the minimum cost needed to reach the desired block from the left. This part covers rightward movement.

3. Minimum cost to go diagonal(rr[i-1][j-1]):
   Here, we find the minimum cost needed to reach the desired block by coming diagonally from the top-left. This part covers diagonal movement.

b) Write the recurrence relation for subproblems.
Ans:
If rr is the minimum cost to reach the desired block:

$$rr[i][j] = \min(rr[i-1][j], rr[i][j-1], rr[i-1][j-1]) + fee(i,j)$$

4. **Assume we have N workers. Each worker is assigned to work at one of M factories. For each of the M factories, they will produce a different profit depending on how many workers are assigned to that factory. We will denote the profits of factory i with j workers by P(i,j). Develop a dynamic programming solution to find the assignment of workers to factories that produce the maximum profit.(Mention the pseudocode).**

Ans:
Algorithm:
Here, we have to use for loop to iterate through the number of factories. Another for loop iterates through the number of workers.
Third nested loop has the recurrence relation to get the maximum profit between the current factory's profit and the optimal assignment of workers from the previous factories. This will give us the maximum profit.
Now, backtrack through the table to find the assignment of workers to factories that yield maximum profit.
If the current assignment of workers to the current factory contributes to the maximum profit, update the list with factory and worker index.
Display the maximum profit and the optimal assignment of workers to factories.

Pseudo Code: -

```
def question_four(N, M, profit):
   for a in range(M + 1):
         rr = [0] * (N + 1)  #Initialize the recurrence relation

   #Find the recurrence relation value
   for i in range(1, M + 1):
      for j in range(N + 1):
         for k in range(min(j, len(profit[i-1])) + 1):
            rr[i][j] = max(rr[i][j], rr[i-1][j-k] + sum(profit[i-1][:k]))

  #Maximum Profit at the last element of recurrence relation
   max_profit = rr[M][N]
   assignment = []
   i, j = M, N

  #Backtrack to find the optimal assignment of workers to factories
   while i > 0 and j > 0:
      for k in range(min(j, len(profits[i-1])), -1, -1):
         if rr[i][j] == rr[i-1][j-k] + sum(profit[i-1][:k]):
            assignment.append((i, k))
            j -= k
            i -= 1
            break

# Display the maximum profit and the optimal assignment of workers to factories.
   return max_profit, assignment[::-1]
```

5. **You have two rooms to rent out. There are n customers interested in renting the rooms. The ith customer wishes to rent one room (either room you have) for d[i] days and is willing to pay bid[i] for the entire stay. Customer requests are non-negotiable in that they would not be willing to rent for a shorter or longer duration. Devise a dynamic programming algorithm to determine the maximum profit that you can make from the customers over a period of D days.**

   Ans:

   Algorithm:

   Here, we have to find the maximum profit that you can make from the customers in D days. For that, we initiate a for loop iterating through number of customers. In this loop, we initialize a nested for loop iterating through number of days. In this loop, we create a recurrent relation such that it finds the maximum value of the maximum profit obtained from i^{th} customer in j days, and maximum profit with current customer.

   Time complexity:

   Since there are 2 nested loops, one iterating in n time and other in D, the time complexity is O(n*D).

   a) Define (in plain English) subproblems to be solved.

   Ans:

   Here, there are two subproblems that we have to solve.

   1. Without including the current customer(rr[i-1][j]):
      Here, we find the maximum profit obtained by considering customers up to the previous index of customer using the same number of days.

   2. With the current customer(rr[i-1][j-d[i-1]] + bid[i]):
      Here, we find the maximum profit obtained by the ith customer's bid while accommodating customers up to the (i-1)th customer and using (j - d[i]) days.

   b) Write the recurrence relation for subproblems.

   Ans:

   If rr is the maximum proft made from customers over a period of D days:

   rr[i][j] = max(rr[i-1][j], rr[i-1][j-d[i-1]] + bid[i])

6. **You are given a sequence of n numbers (positive or negative): x .Your job is to select 1 , x 2 ,. . . , $x\ n$ a subset of these numbers of maximum total sum, subject to the constraint that you can't select two elements that are adjacent (that is, if you pick x then you cannot pick either or). On $i\ x\ i$–1 $x\ i$ the boundaries, if x is chosen, cannot be chosen; if is chosen, then cannot be chosen. 1 x 2 $x\ n\ x\ n$–1 Give a dynamic programming solution to find, in time polynomial in n, the subset of maximum total sum.**

Ans:
Explanation:
Here, we have to find a subset of maximum total sum. For this, we initialize a for loop iterating through the sequence. In this loop, we initiate the recurrence relation to find the maximum total sum.

Algorithm:
Create an array to store the maximum total sum.
Initiate a for loop. In this loop, create a recurrent relation to find the maximum value of among the maximum total sum of the sequence ending in previous index, and the maximum total sum of the sequence ending in index that is 2 values behind the current index + sequence till current index.
Print the latest element in the recurrence relation.

Time complexity:
Here, we have one for loop iterating n times. Hence, the time complexity is O(n).

7. **Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth pi dollars. Devise a Dynamic Programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.**
   Ans:
   Explanation:
   Here, we try to find the optimal way to cut the rod to maximize the total value of the cut pieces. We create a list to store the maximum revenues for various rod lengths. Then we initialize a for loop for rod length from 1 to total length (to check all possibilities). In this loop, we go through all possible cuts and find the maximum revenue recursively. The final element in the list would be the maximum revenue for total rod length.

   Algorithm:
   Create a list to get the maximum revenues for various rod length (Initially 0).
   Initialize a for loop from length 1 to total length.
   In this loop, create a nested loop to iterate through all possible lengths of the cut pieces for rod of various length. Find the max value of revenue among all possible cuts. Store it in the list.
   End both for loop
   Print the last element of the list.

   Time Complexity:
   Here, we have 2 nested for loops iterating in n^2 time. Hence, the time complexity is $O(n^2)$.