Homework 7

aryanrao

1. To utilize dynamic programming for this problem, we'll employ a 2D array named "dp." Here, dp[i][j] indicates whether it's feasible to obtain a sum of j using the first i elements from set A.

   Recurrence relation: dp[i][j] = dp[i-1][j] || dp[i-1][j-A[i]]
   To clarify, dp[i][j] is set to true if we can achieve a sum of j using the first i elements of set A. There are two cases to consider: either we include the ith element, A[i], in the subset, or we exclude it. If we exclude the ith element, the outcome depends on whether we can obtain a sum of j using the i-1 elements, which is represented by dp[i-1][j]. On the other hand, if we include the ith element, we need to check whether we can achieve a sum of j – A[i] using the first i-1 elements, which is represented as dp[i-1][j-A[i]].

   Pseudocode:

```
def isSubsetHalfSumPossible(arr):
n = len(arr)
sum = sum(arr)


    # If the total sum is odd, it is not possible to find a subset with half sum.
    # If we can use decimal points as well, then this if statement can be removed, and the code will run fine.
    if total_sum % 2 == 1:
        return False

    t_sum = total_sum // 2  # Half of the total sum

    # Declare and initialize a Boolean 2D array to store dp results.
    dp = [[False for _ in range(t_sum + 1)] for _ in range(n + 1)]

    for i in range(1, n):
        dp[i][0] = True

    # Fill dp using recursion relation
    for i in range(1, n):
        for j in range(1, t_sum + 1):
```

```
            if j >= arr[i - 1]:
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - arr[i - 1]]
            else:
                dp[i][j] = dp[i - 1][j]

    # Results are stored in dp[n][t_sum]
    return dp[n][t_sum]
```

Correctness:

We begin by initializing a 2D Boolean array called "dp," with dimensions (n+1) * (sum + 1), where n represents the number of elements in set A, and sum denotes the sum of all elements in set A. To ensure the correctness of our approach, we set dp[i][0] to true for all i because obtaining a sum of 0 is always possible with an empty subset. Additionally, we set dp[0][j] to false for all j > 0 since we cannot achieve a sum greater than 0 using an empty set, and since we are dealing with non-negative integers.

We then proceed to populate the dp array using a nested loop, adhering to the recursive relation. Finally, the result will be stored in dp[n][sum/2], which will determine whether it is possible to obtain a sum of sum/2 using all elements from set A. Through this method, we effectively avoid recursion and solely harness the power of dynamic programming, resulting in an efficient solution.

Time complexity:

The process of filling the 2D array **dp** takes O(n * s) time, where 'n' represents the number of elements in set 'A', and 's' denotes the sum of all elements in 'A', which is the dominant factor in this algorithm. Therefore, the overall time complexity of the algorithm is O(n * s).

2. To solve this problem using dynamic programming, we can define a 2D DP array to store the maximum profit achievable at each cell of the maze. To adapt this into a non-recursive dynamic programming solution, we'll iteratively fill in a matrix `DP`, where `DP[i][j]` is the maximum profit possible when arriving at cell `(i, j)`.

   Recurrence Relation:

   Let **maze** be the input 2D array representing the maze with diamond values and costs of moving. Let **dp[i][j]** represent the maximum profit that can be obtained by reaching cell (i, j). Then the recurrence relation can be defined as follows:

   Explanation:

   The recurrence relation is based on the following observation: To maximize the profit at cell (i, j), we need to choose the optimal path that gives us the maximum profit considering the three possible moves: right, down, and diagonal. We add the value of the diamond in the current cell **maze[i][j]** to the maximum profit that can be obtained from the three adjacent cells (left, top, and diagonal) while taking into account the costs of moving (2 for right or down, and 3 for diagonal).

   dp[i][j] = maze[i][j] + max(dp[i][j-1] - 2, dp[i-1][j] - 2, dp[i-1][j-1] - 3)

   Pseudocode:

```
function findMaxProfit(maze):
    M = maze.length
    N = maze[0].length

    // Create a 2D array dp to store the maximum profit at each cell
    dp = new 2D array of size (M+1) x (N+1)

    // Initialize dp array with negative infinity to cover edge cases
    for i from 1 to M:
        for j from 1 to N:
            dp[i][j] = float('-inf')

    dp[1][1] = maze[1][1]

    for i from 1 to M:
        for j from 1 to N:
```

```
        if i != 1 or j != 1:  // Skip the top-left cell as it's already initialized
            // Calculate the profit gained from the left, top, and diagonal
cells
            fromLeft = dp[i][j-1] - 2 if j > 1 else float('-inf')
            fromTop = dp[i-1][j] - 2 if i > 1 else float('-inf')
            fromDiagonal = dp[i-1][j-1] - 3 if i > 1 and j > 1 else float('-inf')

            // Calculate the maximum profit to reach the current cell
            dp[i][j] = maze[i][j] + max(fromLeft, fromTop, fromDiagonal)

    return dp[M][N]
```

Time Complexity:

The time complexity of the algorithm is O(M * N), where M is the number of rows and N is the number of columns in the maze. The two nested loops iterate through all the cells in the maze once to calculate the maximum profit for each cell, making it a quadratic time complexity.

3. This algorithm is based on the Bellman-Ford algorithm, which is commonly used to find the shortest path from a given source vertex to all other vertices in a graph with positive or negative edge weights. We modify the Bellman-Ford algorithm to additionally count the number of shortest paths from the source to each vertex.

   The **distances** array keeps track of the shortest distances from the source to all vertices, and the **pathCount** array keeps track of the number of shortest paths from the source to all vertices. Initially, we set the distance of the source vertex to 0 and the path count to 1 since there is one path from the source to itself.

   The algorithm then iterates **number_of_vertices - 1** times to find the shortest distances and count the shortest paths. In each iteration, it relaxes all the edges (u, v) in the graph by checking if the distance to v can be reduced by going through u. If a shorter path is found (i.e., **distances[u] + weight(u, v) < distances[v]**), we update the distance and the path count for vertex v.

   If the distance to v remains the same (i.e., **distances[u] + weight(u, v) == distances[v]**), it means there is an alternative path to v with the same distance as the currently known shortest path. In this case, we add the path count from u to the path count of v since both paths are part of the shortest paths.

   Finally, the algorithm returns the path count for the destination vertex, which represents the number of shortest paths from the source to the destination.

   Pseudocode:

```
function shortestPathsCount(graph, source, destination):
    distances[source] = 0
    pathCount[source] = 1

    for each vertex v in graph:
        if v != source:
            distances[v] = INFINITY
            pathCount[v] = 0

    for i from 1 to number_of_vertices - 1:
        for each edge (u, v) in graph:
            if distances[u] + weight(u, v) < distances[v]:
```

```
            distances[v] = distances[u] + weight(u, v)
            pathCount[v] = pathCount[u]
        else if distances[u] + weight(u, v) == distances[v]:
            pathCount[v] = pathCount[v] + pathCount[u]

    return pathCount[destination]
```

Correctness:

The algorithm guarantees correctness because it is based on the Bellman-Ford algorithm, which is proven to find the correct shortest paths in a graph with no negative weight cycles. By keeping track of the path counts during the relaxation process, we ensure that we count the number of shortest paths accurately.

Time Complexity:

The runtime complexity of the algorithm is O(V * E), where V is the number of vertices, and E is the number of edges in the graph.

The outer loop runs for V-1 iterations, and the inner loop runs for E iterations, as it iterates through all the edges. In each iteration of the inner loop, we perform constant time operations, like updating distances and path counts. Therefore, the total number of operations in the algorithm is proportional to V * E.

The Bellman-Ford algorithm runs in O(V * E) time, and since our algorithm is essentially a modification of the Bellman-Ford algorithm with additional constant-time operations, its overall runtime complexity remains O(V * E).

Justification of Efficiency: Although the algorithm's runtime complexity is O(V * E), it is still efficient for most practical graphs. The Bellman-Ford algorithm is known to handle graphs with negative edge weights, which many other shortest path algorithms cannot handle. Furthermore, the algorithm's ability to count the number of shortest paths from the source to a given destination adds a useful functionality that is not present in many other shortest path algorithms. Therefore, the algorithm is a reasonable choice when both finding the shortest paths and counting their number is required.