1 Problem 1 (60 points).

Consider a directed graph arranged into rows and columns. Each vertex is labeled (j, k), where row j lies between 0 and M , and column k lies between 0 and N . The start vertex is $(0, 0)$ and the destination vertex is (M, N). Each vertex (j, k) has an associated weight $W(j, k)$. There are edges from each vertex (j, k) to at most three other vertices: $(j + 1, k)$, $(j, k + 1)$, and $(j + 1, k + 1)$, provided that these other vertices exist. That is, the input of your algorithm is a 2-d array $W[0...M, 0...N]$, where each array element $W[i, j]$ represents the weight of the node at position (i, j).

Find a path from $(0, 0)$ to (M, N) that minimizes the total weight (the sum of the weights of all vertices along the path).

1. Provide a recursive formula that computes the exact value that is specified in the problem.

   A recursive formula to brute-force the solution would be to recursively go through each direction possible, starting from right for example, then diagonal, then down, checking every possible path and returning the smallest sum of the weights that the recursion finds along with the path that the recursion took.

2. Express the recursive formula as a bottom-up table-driven dynamic programming algorithm and determine its running time.

   Create a 2D array the same size as the graph. This array will represent the minimum weight needed to reach each vertex. Initialize all values to infinity, meaning each initial minimum path is going to be the shortest found at the time. Starting from the destination vertex (M,N), traverse to all adjacent vertices, adding each of the adjacent vertices' weight to the weight of the destination vertex. Compare each value to the current value of the minimum weight needed to get to that vertex so far, and if it is smaller, replace it with the new minimum. After a vertex has searched all its adjacent paths, set a "searched" flag. Once done for the first level, do the same thing with each of the ones that were just checked, adding the weight of each adjacent vertex to the minimum value of the adjacent vertices and storing that into the vertex being checked if it is smaller than the current minimum. This should only be done when a vertex's "searched" flag is NOT set. However, finding a smaller minimum weight for a vertex should ignore the searched flag, and continue iterating through the paths again, updating any potential smaller minimum weights after the update. This strategy prevents any previously searched paths from being searched again when it is obviously never going to find smaller values. Continue this until each path stops at vertex (0,0), which is the destination, and every array slot should have a value for minimum weight paths.

   The running time of this algorithm is very dependent on case by case situations. Best case scenario only searches through each edge roughly one time, while worst case sce-

narios have to search each edge 3*(M+N) times, 3 representing the number of paths that can influence the amount of times an edge has to recalculate paths, and M+N being the longest path that exists in the graph. So, the running time of this algorithm is O(E*(N+M)), E being the number of edges.

3. Write an algorithm that uses the values stored in The table to find an optimum path, and determine its running time.

   The algorithm for interpreting the actual path is much simpler. The algorithm starts at (0,0) and checks the value in the minimum weight path table. It then checks to see which of its possible paths, after subtracting that node's minimum weight from the value it got before, equals the value of the current node's weight. This will give the correct path because the above algorithm had found the cheapest way to eventually get to the starting vertex, so each of the other vertex's on the path will have a larger value after subtracting the weight of that vertex. The path will then go along that vertex's path that it found to be correct, and repeat the process for the next vertex. Eventually, subtracting the destination vertex weight from the current vertex's minimum weight will give the value of the vertex itself, it travels to the destination vertex, and the path is complete.

   The running time of this algorithm is, at worst, 3(N+M). The 3 represents the maximum number of paths that need to be checked per travel, and the N+M represents the maximum number of travels that need to be done. Therefore, the running time of this algorithm is O(N+M)

2 Problem 2 (40 points).

Search and learn three existing algorithms that use the dynamic programming strategy, in addition to those that we have discussed the class ("rod-cutting" and "matrix-chain"). For each algorithm, in your own language, concisely and clearly describe:

1. The problem statement
2. Why recursion-based idea does not work
3. Why dynamic programming works
4. The source of your finding

2.a Coin Change Problem

1. Given an array of coin values of size n and an integer sum, find all the possible ways to get the sum using different combinations of the coin values.

2. Recursion does not work here because, as the coin values are subtracted from the sum, the recursion is going to come across sub-problems that have already been solved, especially when it gets down to the sub problems of 1 or 2 coin combinations.

3. Dynamic programming works because a 2D array is used to store previously solved sub-problems for later use. This problem and solution is similar to the rod cutting example, but instead of finding the highest "profit", we're instead finding all possible combinations period, given a more restricted "size" (or in this case, value) list

4. `https://www.geeksforgeeks.org/coin-change-dp-7/`

2.b Fibonacci

1. A mathematical sequence of numbers such that each term is the sum of the previous two terms. Find the value of an arbitrary term in the fibonacci sequence.

2. Recursion is very taxing in this example, because not only are sub-problems guaranteed to repeat many, many times due to the n-1 n-2 nature of the sequence, but it is using two separate recursive calls per number to do so. So it is both taxing in time cost and in memory cost.

3. A simple int array for dynamic programming works because each value can be stored in the array as it finds it, given the base cases 0 and 1, so instead of needing to work backwards to 0/1, it can work forwards from 2 to the given index, storing values for reference as it goes, which means the algorithm goes from exponential time and memory costs to literally linear time and memory cost, as no recursion is needed and everything is stored in one array.

4. `https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/`

2.c Minimum Sum Path in Triangle

1. Given a triangle structure of numbers, find the path with the smallest sum of numbers going from top to bottom. Paths must be adjacent to previous number.

2. Recursion is very costly here as two recursive calls are needed per next step that's taken, and given that the triangle is likely stored in a 2D array, extra calls might even be made that have no value in them.

3. Dynamic programming works well here because a bottom-up approach and separate array of values can be used to store the minimum values found as it adds above values. Also, instead of needing recursive function calls, it simply references the values previously stored in the array.

4. `https://www.geeksforgeeks.org/minimum-sum-path-triangle/`