1. Modified Change Problem: Apply the dynamic programming algorithm to find all the solutions to the change-making problem for the denominations 1, 3, 5 and the amount $n = 9$.

```c
int num_coins(int n)
{
    if(n == 0){ return 0; }

    // ~0U is unsigned practical infinity, shift right 1 for two's complement
    if(n < 0) { return ~0U >> 1; }

    if(TABLE[n] == -1)
    {
        for (int i = QUIN; i <= n; ++i)
        {

            int vals[DENOMINATIONS] = { TABLE[i - QUIN]
                                      , TABLE[i - TRIPLE]
                                      , TABLE[i - PENNY]
                                      };
            TABLE[i] =  1 + minimum(vals, DENOMINATIONS);
        }
    }
    return TABLE[n];
}
```

Figure 1: Source code for Tabular make change algorithm

```
chaos2022  ~/CS350/dynamic_programming  master ± ● ↑1
  ./make_change 9 -v

Num Coins: 3

Enumerated Results:
[ 0, 1, 2, 1, 2, 1, 2, 3, 2, 3]
```

Figure 2: Output for the make change program

2. Rod Cutting Problem Design a dynamic programming algorithm for the following problem. Find the maximum total sale price that can be obtained by cutting a rod of n units long in to integer-length pieces if the sale price of a piece i units long is pi for i = 1, 2, . . . , n. What are the time and space efficiencies of your algorithm?

```
#elif defined(TABULAR)

unsigned rod_cutting(unsigned* prices, unsigned length)
{
    int table[length + 1];
    table[0] = 0;

    for(int i = 0; i <= length; ++i)
    {
        int max_price = 0;
        for(int j = 0; j < i; ++j)
        {
            max_price = max(max_price, prices[j] + table[i - j - 1]);
        }
        table[i] = max_price;
    }
    return table[length];
}
```

Figure 3: Source code for tabular rod cutting algorithm

This tabular algorithm has a space complexity of $\Theta(n)$ and a time complexity of $\Theta(n^2)$, where n is the length of the rod.

3. Minimum Sum Descent: Find the smallest sum in a descent from the triangle apex to its base through a sequence of adjacent numbers. Design a dynamic programming algorithm for this problem and indicate its time efficiency.

```c
int min_sum_descent(int* triangle, unsigned rows, unsigned size)
{
    while(rows > 0)
    {
        int child_row  = size - rows;
        int parent_row = size - rows - rows + 1;
        int diff       = child_row - parent_row;

        for(int i = parent_row; i < child_row; ++i)
        {
            int left_child  = triangle[i + diff];
            int right_child = triangle[i + diff + 1];

            triangle[i] += min(left_child, right_child);
        }
        size -= (rows--);
    }
    return triangle[0];
}
```

Figure 4: Source code for tabular minimum sum descent algorithm

This tabular algorithm has a space complexity of $\Theta(n^2)$, since to represent the triangle we need to use at least $\sum_{i=1}^{n} i$ space. Likewise, the time complexity is $\Theta(n^2)$ as well since for each element in each row up to $n-1$ rows we must find the minimum value of the nodes children. I.E. $\sum_{i=1}^{n-1} i$ work is done.

4. N-Choose-K Problem: Design a dynamic programming algorithm for the recursive n-choose-k problem discussed in class and then indicate its time and space complexity.

```
#elif defined(TABULAR)

unsigned min(unsigned a, unsigned b)
{
    return (a < b) ? a : b;
}

unsigned long choose(int n, int k)
{
    if(k > n) { return 0; }

    TABLE[0][0] = 1;
    for(int i = 1; i <= n; ++i)
    {
        for(int j = 0; j <= min(i, k); ++j)
        {
            unsigned long left  = (j - 1 < 0) ? 0 : TABLE[i-1][j-1];
            unsigned long right = (j > i - 1) ? 0 : TABLE[i-1][j];

            TABLE[i][j] = left + right;
        }
    }
    return TABLE[n][k];
}
```

Figure 5: Source code for tabular binomial coefficient algorithm

The time and space complexity for this tabular algorithm are both $\Theta(n \cdot k)$

5. Pebble Collecting Problem: Design a dynamic problem solution for the pebble collecting problem described in class, and indicate its time and space efficiency.

```
#elif defined(TABULAR)

int pebbles(int i, int j, int** graph)
{
    /* Calculates the maximum number of pebbles that can be collected on a
    any lattice path.

    Preconditions:
      graph must be correctly initialized and each element on the graph
      is either a 1 (has a pebble) or a 0 (does not have a pebble).

      i and j are not bounds checked and it is assumed that graph[i][j]
      is a valid entry on the graph.
    */

    if(TABLE[i][j] < 0)
    {
        for(int n = 0; n <= i; ++n)
        {
            for(int m = 0; m <= j; ++m)
            {
                int up   = (n > 0) ? TABLE[n-1][m] : 0;
                int left = (m > 0) ? TABLE[n][m-1] : 0;

                TABLE[n][m] = graph[n][m] + max(up, left);
            }
        }
    }
    return TABLE[i][j];
}
#else
```

Figure 6: Source code for tabular pebble collecting algorithm

The time and space complexity for this tabular algorithm are both $\Theta(n \cdot m)$ where n and m are the dimensions of the graph.