Algorithm 1 (with the entire table):

This algorithm is like the pseudocode given by Eric Vagoda in the assigned videos and readings. With inputs of cap capacity, n items, and a weight weights[i] and value values[i] for each item, the algorithm constructs a dynamic programming table of size (n+1)x(cap+1). It will one-index the w and v arrays to make table look-up seamless. It initializes the first row and the first column of the table as 0. With an outer loop of n iterations and an inner loop of W iterations, it will traverse the table and assign values. It looks like this:

```
table = [[0 for x in range(cap+1)] for x in range(n+1)]     #[[cols]rows]
for i in range(1, n+1):
    for b in range(1, cap+1):
        if weights[i] <= b:
            table[i][b] = max(values[i]+table[i-1][b-weights[i]], table[i-1][b])
        else:
            table[i][b] = table[i-1][b]

bestVal = table[n][cap]
```

Because the full table is stored in memory, it has a memory complexity of O(nW). The time complexity is also O(nW) due to the nested for loops. All table lookups and comparisons are done in constant time, so they are ignored.

Algorithm 2 (using one row of the table at a time):

This algorithm does the same thing as above, except that since when we are looking for the optimal solution, you only really consider the row directly above the row you are currently filling, we will not store the entire table. Instead, by just storing the above row and the current row, the memory complexity greatly improves to O(W), since there are W+1 entries per row, and we only store AT MAX 2 full rows at any point. After the current row is filled up, the above row is overwritten by the current and the current is reinitialized to empty.

```
#KNAPSACK OPTIMAL VALUE STORING ONLY ONE FULL ROW AND THE CURRENT ROW
above = [0 for x in range(cap+1)]       #this one stores the row above
for i in range(1, n+1):
    current = [0 for x in range(cap+1)]              #this stores the current row
    for b in range(1, cap+1):
        if weights[i] <= b:
            current[b] = max(values[i]+above[b-weights[i]], above[b])
        else:
            current[b] = above[b]
    above = current             #above is overwritten at the end of filling up
current

bestVal = above[cap]
```

As mentioned above, the memory complexity goes to O(W). However, the time complexity still sits at O(nW) due to the fact that nested for loops are still being used. We still must iterate through each entry, so the outer loop is still O(n) and the inner loop is O(W). All else is done in constant time, so we can ignore it.

OPTIMAL VALUES OF THE SOLUTIONS:

My code outputs the VALUE of the optimal solution to the terminal for the specified input. The input can easily be changed by hardcoding it. By default, I have submitted the code that deals with the "small.txt" input file due to quick execution cycle of the program.

```
C:\Users\camli\Documents\Spring 2020\CSC 222\Program2a>python knapsack.py
Using a 2d table: 19
Storing only one row: 19

C:\Users\camli\Documents\Spring 2020\CSC 222\Program2a>python knapsack.py
Using a 2d table: 632
Storing only one row: 632

C:\Users\camli\Documents\Spring 2020\CSC 222\Program2a>python knapsack.py
Using a 2d table: 55636
Storing only one row: 55636
```

using small.txt

using medium.txt

using large.txt