# CSC 222 Programming Assignment 2

Andrew Knox

March 31st, 2020

```
#Function to build DP table, using O(n*W) memory, O(n*W) time
def buildTableNW(w, j, dfT, dfData):

    #Return -maxVal so that the max() function always
    #picks K(w, j-1) when w(j) > w
    maxVal = dfData.loc[:, "Value"].max()
    if w < 0:
        return (maxVal * -1)

    #If dfT(w, j) has not been filled in, fill it in with equation
    if dfT.iloc[w, j] == -1:
        dfT.iloc[w, j] = max(buildTableNW(w, j-1, dfT, dfData),
                             (buildTableNW(w - dfData.loc[j, 'Weight'],
                              j-1, dfT, dfData) + dfData.loc[j, 'Value']))
        return dfT.iloc[w, j]

    return dfT.iloc[w, j]
```

This is my function for problem a(1). The optimal values for small and medium were 19 and 632, respectively. I couldn't get large to work, it was too large for my function a(1), and a(2) took too long.

a(1) function: The maximum recursion that the function will do is O(nW), running time is O(nW). Memory usage is O(recursion depth * nW).

```
#Helper method that loops through all columns
def buildTableWHelper(w, j, dfT, dfData):
    for i in range(1, dfT.shape[1]):
        dfT.iloc[:, i] = buildTableW(w, i, dfT.iloc[:, i-1], dfT.iloc[:, i], dfData)
    return dfT.iloc[w, j]



#Function to build DP table, using O(W) memory, O(n*W) time
```

```
def buildTableW(w, j, firstCol, secondCol, dfData):

    #Fills in secondCol, then uses it as next firstCol
    for i in range(firstCol.size - 1, 0, -1):
        newWeight = i - dfData.loc[j, 'Weight']
        if newWeight < 0:
            secondCol.iloc[i] = firstCol.iloc[i]
        else:
            secondCol.iloc[i] = max(firstCol.iloc[i],
                                    firstCol.iloc[newWeight]+dfData.loc[j, 'Value'])
    return secondCol
```

This is my function for problem a(2). The running time is O(nW), because the for loop in the helper method runs n times, and the non-helper method has a for loop that goes W times. Everything inside these loops is constant.
The memory usage is O(W) because although buildTableWHelper() uses O(nW) memory, the buildTableW() function only uses 2 columns, meaning the memory complexity will be O(2*W), or O(W).

```
#Function to backtrack given DP table
def backtrackTableSlow(dfT, dfData, W, n):
    optimalValue = dfT.iloc[W, n]
    solutionList = []
    while W > 0:
        if dfT.iloc[W, n] == dfT.iloc[W, n-1] or dfT.iloc[W, n] == -1:
            n = n - 1
        else:
            W = W - dfData.loc[n, 'Weight']
            solutionList.append(n)
            n = n - 1
    fileSize = input("What size file was this? ")
    f = open(fileSize + "-output.txt", "w")
    f.write("V " + str(optimalValue) + "\n")
    f.write("i " + str(len(solutionList)) + "\n")
    for i in range(len(solutionList) - 1, -1, -1):
        f.write(str(solutionList[i]) + "\n")

backtrackTableSlow(dfT, dfData, totalWeight, numItems)
```

This is my backtracking function. It either takes 1 step to the left and no steps up, or 1 step to the left and a step up from w to w - $w_j$. It has to loop until n is 0, so it will step a total of O(n) times.

By the way, I didn't notice that the backtracking was part of part b. Oops.