CSC 222 Programming Assignment 2b

Andrew Knox

April 2020

```
def partA(totalWeight, numItems, dfData):
    print("This is part A\n")
    #Fill table with O(W + 1, n + 1) items
    filler_data = np.full(shape=(totalWeight + 1, numItems + 1), fill_value=-1)
    dfT = pd.DataFrame(data=filler_data)
    dfT.iloc[0:dfT.shape[0], 0] = 0
    dfT.iloc[0, 0:dfT.shape[1]] = 0

    print("The optimal value is ", buildTableNW(totalWeight, numItems, dfT, dfData), ",
    backtrackTableSlow(dfT, dfData, totalWeight, numItems)

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:
        #print(dfT, "\n")
        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, dfDa
        #print(dfT, "\n")
        return dfT.iloc[w, j]
    #print(dfT, "\n")
    return dfT.iloc[w, j]

#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")
```

This is all of 2a. The time required is O(nW).

```
def partB(totalWeight, numItems, npData, dfData):
    print("This is part B\n")
    #print("The optimal value is:", npTableLowMem(totalWeight, numItems, npData), "using

    optimalVal, midIndex = lowMemBacktrack(totalWeight, numItems, npData)
    print("Optimal value is", optimalVal, "and backtracking crosses column n//2 at row i

    #findPath(0, 0, totalWeight, numItems, npData)

#Part 2bi
def lowMemBacktrack(totalWeight, numItems, npData):
    #fill first column (j==0)
    Kprev = np.full(shape=(totalWeight + 1), fill_value=0)
    Mprev = np.full(shape=(totalWeight + 1), fill_value=0)
    #fill second column (j==1)
    Kcurr = np.full(shape=(totalWeight + 1), fill_value=-1)
    Mcurr = np.full(shape=(totalWeight + 1), fill_value=-1)
    Kcurr[0] = 0

    #booleans for backtracking
    passedHalf = False
    firstHalfPass = True

    #Fill in second column based on what's in first column
    for j in range(1, numItems + 1):
        for w in range(0, totalWeight + 1):
            #CASE 1: w == 0
            if w == 0:
                Kcurr[w] = 0
                continue
            currentWeight = npData[j-1, 0]
```

```
            #CASE 2: w < weight(j), return K(w, j-1)
            if w < currentWeight:
                Kcurr[w] = Kprev[w]

            #CASE 3: K(w, j) = max(K(w, j-1), K(w-weight(j), j-1) + value(j))
            else:
                firstVal = Kprev[w]
                secondVal = Kprev[w - currentWeight] + npData[j-1, 1]
                if firstVal > secondVal:
                    Kcurr[w] = firstVal
                else:
                    Kcurr[w] = secondVal

        #First time passing half, start backtracking
        if j > numItems // 2 and firstHalfPass:
            passedHalf = True
            firstHalfPass = False
            Mprev = np.arange(0, totalWeight + 1)
            Mcurr = np.zeros(totalWeight + 1)

        #Backtracking work
        if j > numItems // 2:
            for w in range(0, totalWeight + 1):
                if Kcurr[w] == Kprev[w]:
                    Mcurr[w] = Mprev[w]
                else:
                    currentWeight = npData[j-1, 0]
                    Mcurr[w] = Mprev[w - currentWeight]

        #If j is not the last column, make Kcurr into Kprev and create new Kcurr
        if j != numItems:
            #print(Kcurr.values)
            Kprev = Kcurr
            Kcurr = np.full(shape=(totalWeight + 1), fill_value=-1)


        if j > numItems // 2 and j != numItems:
            Mprev = Mcurr
            Mcurr = np.full(shape=(totalWeight + 1), fill_value=-1)

    #Return optimal value and row index when backtracking path reaches item n//2
    return (Kcurr[totalWeight], Mcurr[totalWeight])

#Part 2bii
#def findPath(startRow, startCol, endRow, endCol, npData):
#    solutionList = []
```

```
#    answer = (0, 0)
#    #BASE CASE
#    if endCol - 1 == startCol:
#        return (startRow, startCol)

#    optimalVal, kVal = lowMemBacktrack(endRow - startRow, endCol - startCol, npData)
#    kVal += startRow

#    answer = findPath(startRow, startCol, kVal, endCol//2, npData)
#    solutionList.append(answer)
#    answer = findPath(kVal, endCol // 2, endRow, endCol, npData)
#    solutionList.append(answer)

#    print("Now we must combine")
```

This is all of part b, including the code that I ALMOST got working for part ii. The algorithm that gets the middle index is O(W) memory because it creates the same amount of space as the O(W) algorithm that computes the optimal value. So in total the memory usage would be O(2W), or O(W).
For part ii, I almost got it working, but I can still prove it's only O(W) memory. Every time you call the find-path function, you must compute a new middle-value for all items larger than the floor of n/2. Thus you're calling a method that takes O(W) memory once, then O(W/2) memory, and so on. So it ends up becoming O(2W), or O(W).