Name: Can
Surname: Gök
Student Id: 150118014

Algorithm of Analysis, Homework 2 Report

Multi Constraint Knapsack Problem

Algorithm

The algorithm on that project works with multiple variables: weights and values of the items. These values are the main focus to calculate the maximum value that knapsacks can get.

This project includes three different approaches: **distance approach**, **weight approach** and **value efficiency approach**.

The accuracy for distance and weight approaches are really low and that's why they are not going to be introduced in this report. But their functions can be reached via python code. (They are not connected to the main program.)

Value efficiency approach is the method of this application.

Value Efficiency Approach

```
sortedValueArray = [[]]
def calculateValueEfficiency(array, length):
        global sortedValueArray
        sortedValueArray.clear()
        sortedValueArray = [[]]
        i = 0
        value = 0
        for i in range(0, length):
                if(array[i][1] != 0) : value = n_values[array[i][0]] / array[i][1]
                else: value = n_values[i]
                sortedValueArray[i].append(array[i][0])
                sortedValueArray[i].append(value)
                if(i < length - 1): sortedValueArray.append([])
        sortedValueArray.sort(key=takeSecond, reverse=True)
        return sortedValueArray
```

Above code is the algorithm of the calculate value efficiency function. Value efficiency includes two main variables: **Weight** and **Value** of the items. The function calculates every item's summed item weights and divides the value of the item with this sum. This gives us the efficiency of every variable.

The algorithm creates a new multi dimensional array that includes efficiency of every item and sorts the list with their efficiency.

Example efficiency array with 3 elements, created output for test_4.txt:

```
[[19, 0.018614039022202288],
 [25, 0.018384131591678763],
 [26, 0.016131634134537828]]
```

The algorithm later looks for every item in a decreasing order. When one of the knapsacks is filled, the remaining items that have a weight higher than the empty weight of the filled knapsack are removed from the efficiency array and the algorithm calculates the knapsacks again until there are no items left in the efficiency array.

```
weightArray = [[]]
def calculateLowerWeight(value, weightIndexes):
        global weightArray
        weightArray.clear()
        weightArray = [[]]
        i = 0
        newArray = True
        for weightIndex in weightIndexes:
                isHigher = False
                for index in range(0, m):
                        if(mxn[index][weightIndex] >= value):
                                isHigher = True
                if(isHigher == False):
                        if(newArray == False): weightArray.append([])
                        newArray = False
                        weightArray[i].append(weightIndex)
                        weightArray[i].append(mxn[0][weightIndex] + mxn[1][weightIndex])
                        i += 1
        return weightArray
```

The algorithm writes the variables in a binary format to the output file when there are no items left in the efficiency array and finishes the program.

Time Complexity

The time complexity for the function can be calculated as,

n for every efficiency calculation
n for every lower weight calculation

The outer loop's turn count is dependent on the data given to the program. But it has approximately between 0 to **log n** time complexity. Which makes time complexity between n and **n log n** but most likely **n**. **O(n log n)**

How Points Given

```
indexPointArray = [[]]
def givePoints():
        global indexPointArray
        indexPointArray.clear()
        indexPointArray = [[]]
        i = 0
        count = 0
        for i in range(0, n):
                for valueItem in sortedValueArray:
                        if(valueItem[0] == i):
                                indexPointArray[count].append(valueItem[0])
                                indexPointArray[count].append(valueItem[1])
                                count += 1
                                if(count < len(sortedValueArray)): indexPointArray.append([])
                                break
        indexPointArray.sort(key=takeSecond, reverse=True)
        return indexPointArray
```

Give points algorithm finds the indexes of the items and attaches their efficiency value to them which makes that function easily changeable.

Why Value Efficiency Approach?

Value efficiency approach creates a great opportunity and time complexity for solving the knapsack problem. This algorithm easily lowers the time complexity between **n** and **n log n**.