Emirhan UZUN
171044019

CSE 321 - INTRODUCTION TO ALGORITHMS
HOMEWOR #5

1- Firstly, I divided arrays into 2 parts which are bigger than 0 and smaller than 0. Then I sum these values seperately. I created a map to hold values. These operations are done in my helper function. After that, I send these arguments to the main function. In that, if the target sum (for our homework is 0), less than all negative numbers or target sum is bigger than all positive numbers, return False. For base case if the index is equal to zero; return the first element of not zero list (bigger than 0 and smaller than 0 arrays concatenate) is equal to target sum which is 0.

After that, we use our map. If the same value of index and target sum are in map, we return this map value. So we don't search again and again. That is the dynamic programming approach. Also if the result is true, I print the path that sum of values is 0.

NOTE!: The dynamic programming's idea is to simplify store the results of subproblems so we don't have to recompute the values. So it is mainly an optimization over plain recursion. In this question, I used dynamic programming with recursion. I didn't increase the time so much because as we did in normal I used memorized values even I used recursion.

2- Firstly, I create array with length of set. After that, I create 2 different array for print the path which has minimum value. For the bottom row, the program turns length of element number of bottom row. Then I have 2 loop (inner) to find the minimum solution. At the end of method, it returns minimum sum and path. The time complexity of this program is max (O(n), O(n2)) which is n is length of triangle set. O(n) means for bottom row I have 1 loop. After that I have 2 inner loop and these runs n times each of them. So it turns n^2 time. The complexity is $O(n^2)$.

- Code Explanations:

```
n ← length of set n-1
for i in range(len(set[n])):
    totalsum[i] ← set[n][i]
    previousPath[i] ← [set[n][i]]
```

This block fills the arrays which I explained it above (Bottom row)

```
for i in range (len(set)-2, -1, -1)    ⇒ // from length of set minus 2 to -1
  for j in range (len(set[i]))          ⇒ Turns length of ith index of set
    if totalsum[j] > totalsum[j+1]:
        path[j] ← previousPath[j+1] + [set[i][j]]   ⇒ // Assign new path j+1th index of prevPath
    else
        path[j] ← previousPath[j] + [set[i][j]]     ⇒ // Assign new path to jth index of prevPath
    previousPath ← path
    totalSum[j] ← set[i][j] + min (totalSum[j], totalSum[j+1])
return totalSum[0], path[0]
```

It controls every path with saved values. So it is optimization of recursive codes. At the end, returns the minimum value of path and path values.

3- In this algorithm, we have values array, weights array and W. First, I create 2D array store values. The rows are weights and columns are weights in range 1 to W. The array is filled with maximum value and weight according to given arrays. At the end of method, it returns the maximum value that can be put in a capacity of W. I have 2 loops. The outer loop turns n+1 time and the inner loop turns W+1 times. So the complexity of these algorithm is O(n x W).

Code Explanations

```
def knapSack (W, weight[], values [], n(length of values))
    kTable[][] <- (W+1) (n+1) size
    for i in range (n+1)                    // Outer loop
        for w in range (W+1)                // Inner loop
            if i or w equals 0 do           // if i or w is 0, then assign
                ktable [i] [w] = 0          // 0 to table for these index.
            else if weight[i-1] <= w do
                kTable[i][w] = max (val [i-1]+kTable[i-1] [w-weight [i-1]], kTable [i-1][w])
                    # // Assign max of these values to the table.
            else
                kTable [i][w] = kTable [i-1][w]   // Assign same value to these
                                                  index.
    return kTable [n][W]
```