CSE 321

Homework 5 Solutions

1 – To solve this question using dynamic programming;

First i created **dp_table** for the number of sets that have zero sum then created **visited_points** table which is keeps the visited number.In my algorithm, i keep each sub arrays in **sub_list**, in **mem** variable i keep old version of **sub_list** because i used after in recursive calls. Then i append next value to **sub_list** and insert sum of these to **dp_table.**

For dp_table, call the function with adding next element, and without next element to find all possible sub sets.If function paramether index, equals the length of the input algorithm append the sub_list to **sumOfzero** (result array).I used visited table for check index is visited or not, if it is visited then returns the last element of db_table.

The following code, returns all sub sets with total sum of elements equal to zero.

```
procedure sumset_with_dp(input_list):
    sub_list,subSets = [],[]
    find_all_Subsets(subSets,0, 0, input_list, sub_list)
    return subSets
end
```

```
procedure find_all_Subsets(subSets,index, sumOfzero, input_list, sub_list) :
    if (index == len(input_list)) :
        if (sumOfzero == 0):
            subSets.append(sub_list)
            return True
        else :
            return False
    if (visited_points[index][sumOfzero + len(visited_points)]) :
        return dp_table[index][sumOfzero + len(visited_points)]
    else:
        visited_points.append(1);
    mem = sub_list.copy()
    sub_list.append(input_list[index])
    dp_table.append(find_all_Subsets(subSets,index + 1, sumOfzero + input_list[index], input_list, sub_list)
                    +find_all_Subsets(subSets,index + 1, sumOfzero, input_list,mem))
    return dp_table[index][sumOfzero + len(visited_points)]
end
```

Time complexity of algorithm;

$T(n) = O(n*m)$, where n is number of elements of array and m is sum of the elements of array, because in recursive function calling n times and there are m sub problem.

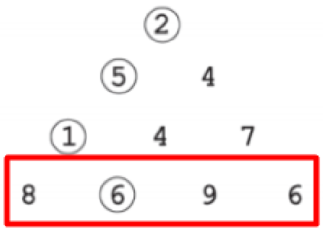
```
INPUT :  [2, 3, -5, -8, 6, -1]
Result 1 = [2, 3, -5]
Result 2 = [2, -8, 6]
Result 3 = [3, -8, 6, -1]
Result 4 = [-5, 6, -1]
```

2 – To solve this problem using dynamic programming, i create a 1d array to remember the min cost row.In first step i fill the 1d array last row of triangle then calculate min cost path each row bottom to top.Each row min cost in the 0^{th} element of 1d array.

In example; [(2), (5,4), (1,4,7), (8,6,9,6)]

The bottom row (step 1 or row 4) is the base case where the path sums for leaf nodes are the values themselves. I will calculate our way from the bottom row to the top.

Step(Row)\index	0	1	2	3
Step4				
Step3				
Step2				
Step1	8	**6**	9	6



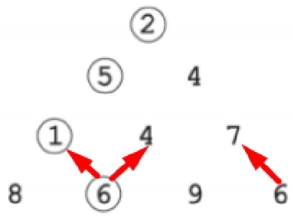
Step 2 index 0 = min (8,6) + 1 = 7

Step 2 index 1 = min (6,9) + 4 = 10

Step 2 index 2 = min (9,6) + 7 = 13

Then step 2 in table ;

Step(Row)\index	0	1	2	3
Step4				
Step3				
Step2	**7**	10	13	-
Step1	8	**6**	9	6

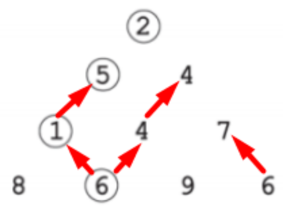


Step 3 index 0 = min (7,10) + 5 = 12

Step 3 index 1 = min (10,13) + 4 = 14

Then step 3 in table ;

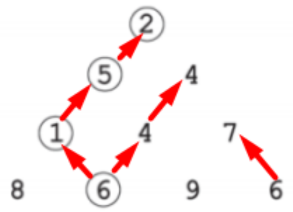
Step(Row)\index	0	1	2	3
Step4				
Step3	**12**	14	-	-
Step2	**7**	10	13	-
Step1	8	**6**	9	6



Step 4 index 0 = min (12,14) + 2 = 14

Then step 4 in table ;

Step(Row)\index	0	1	2	3
Step4	14	-	-	-
Step3	12	14	-	-
Step2	7	10	13	-
Step1	8	6	9	6



After running the algorithm, it returns the 0th index for result.

Sum of path = 14;

Time complexity of this algorithm **T(n) = O(n)** where n is number of elements.

But the table consumes **O(n*m)** where m is number of rows.

3- To solve this problem with dynamic programming, i created 1d dp_table with full of zeros and size W+1 (capacity +1).There are 3 element in problem so our solution has 3 steps. For each step, I select the most valuable elements according to the maximum weight that can be taken in that step and calculate the value. When I do this, in the last step, I calculate the maximum value according to the maximum weight that can be taken in the last index.Algorithm returns dp_table[W] = 16.

			Maximum weight that can be taken										
STEPS	**VALUE**	**WEIGHT**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**dp_Table (1d)**
1	**3**	**2**	0	0	3	3	6	6	9	9	12	12	
2	**4**	**4**	0	0	3	3	6	6	9	9	12	12	
3	**10**	**5**	0	0	3	3	6	10	10	13	13	**16**	

```
procedure knapsack(items_count, values, weights,W):
    dp_table = [0 for i in range(W + 1)]
    for i in range(W + 1):
        for j in range(items_count):
            if (weights[j] <= i):
                dp_table[i] = max(dp_table[i], dp_table[i - weights[j]] + values[j])
            end if
        end for
    end for
    return dp_table[W]
end
```

Algorithm Analysis;

As we can take all items multiple number of times, we check all of them(1 to N) for all weights from 0 to W.Hence time complexity of algorithm is ;

T(n) = Θ ((W+1) * N), where N is number of elements of array and W is capacity of knapsack.