GTU Department of Computer Engineering
CSE 321 - Fall 2020
Homework 5 Report

Akif KARTAL
171044098

Q1)

Problem Definition

The problem is to find if **all subsets** of array with total sum of elements equal to zero.

Inputs: arr(Array)

Outputs: All subsets if any

Solution

Since, this problem request a **dynamic programming algorithm** I made my algorithm by using following approach.

To solve this problem, First we need to find all subsets then check each subset such that sum of elements equal to zero. If sum of elements equal to zero of that subset display that subset on the screen.

To find subsets of a set I used a recursive backtracking algorithm. In this algorithm, while searching a subset I am checking two cases, one of them is include current element of array into the subset and try to find a subset. One of them is don't include current element of array into the subset and try to find a subset.

My Code;

```
def checkWithBacktracking(arr,size,subset):
    if(size<0):
        checkSubset(subset)
        return
    # include current element into the subset and call again
    subset.append(arr[size])
    checkWithBacktracking(arr,size-1,subset)
    # don't include current element into the subset and call again
    subset.pop()
    checkWithBacktracking(arr, size - 1, subset)
```

Parameters:
arr : array
size: size of array
subset : a deque from python collection library to keep found subsets and operate them.

checkSubset is a helper method to check sum of given subset.

Complexity Analysis

In this algorithm since we need to check each subset of n element array, our time complexity is;

$$T(n) \in O(n2^n)$$

Note that since this question wants the "display the elements of each subset whenever it finds them", we need to check each subset therefore our time complexity must be exponential.

Test Results

```
-----------------------------------------------------
Test Array:  [2, 3, -5, -8, 6, -1]
1 ) -1, 6, -8, 3,
2 ) -1, 6, -5,
3 ) 6, -8, 2,
4 ) -5, 3, 2,
-----------------------------------------------------
Test Array:  [2, 3, 5]
No subset with total sum of elements equal to zero!
-----------------------------------------------------
Test Array:  [95, 42, 27, 36, 91, 4, -4, 5]
1 ) -4, 4,
-----------------------------------------------------
```

Q2)

Problem Definition

The problem is to find the smallest sum path from the triangle apex to its base through a sequence of adjacent numbers.

Inputs: 2D array(triangle)

Outputs: smallest sum path

Solution

Since, this problem requests a dynamic programming algorithm we will solve this problem with following dynamic programming algorithm.

To solve this problem, First, we need to find an answer for the elements in the last row which is bottom of triangle. Because we know that smallest sum can be found if we start bottom of the triangle. Then, we continue to find row that above the bottom row. In this row we choose the element which is the most appropriate and also adjacent to the row in below as question emphasize. By this process we continue to going on upward direction. And then, when we reach the top row we found the solution.

My code;

```python
def smallest_sum_path(triangle):
    size = len(triangle)
    for i in range(size - 2, -1, -1):
        for j in range(0, i+1):
            if (triangle[i + 1][j] < triangle[i + 1][j + 1]):
                triangle[i][j] = triangle[i][j] + triangle[i+1][j]
            else:
                triangle[i][j] = triangle[i][j] + triangle[i+1][j+1]
    return triangle[0][0]
```

As I mentioned in this code, solution will be found in bottom-up direction.

Test Results

```
-------------------------------------------------------
Test Triangle:  [[2], [5, 4], [1, 4, 7], [8, 6, 9, 6]]
Smallest Sum Path:  14
-------------------------------------------------------
Test Triangle:  [[5], [3, 4], [9, 8, 1], [4, 5, 8, 2]]
Smallest Sum Path:  12
-------------------------------------------------------
```

Complexity Analysis

In this algorithm we visit across each row and each column, and there is n^2 cells in the triangle.

Therefore, our time complexity is;

$$\mathbf{T(n)\ \Theta(n^2)}$$

Q3)

Problem Definition

The problem is to solve knapsack problem such that in this problem we can pick from each item as many as we can.

Inputs: W, n, weights, values

Outputs: total value that can fit the knapsack.

Solution

This problem is a little different than classical knapsack problem and also, require a dynamic programming algorithm therefore we need to think before start.

One of the step of dynamic programming is;

- Find a recursive formulation for the value of the optimal solution.

If we apply this step we will find following recursive formulation for this problem;

Let K[x] be the optimal value for capacity x then,

- ✓ $\mathbf{K[x] = max_i \{ K[x - w_i] + v_i \}}$

Actually, we solve the big part of this question let's use this recurrence in pseudocode.

```
procedure DifferentKnapsack(W, n, w, v)
        K[0] = 0
        for x = 0, …, W+1
                K[x] = 0
                for i = 0, …, n
                        if wᵢ ≤ x
                                K[x] = max{K[x], K[x – wᵢ] + vᵢ}
                        end if
                end for
        end for
        return K[W]
end
```

My Code from pseudocode;

```python
def different_knapsack(W, n, w, v):
    K = [int] * (W+1)
    K[0]=0
    for x in range(0,W+1):
        K[x]=0
        for i in range(0,n):
            if (w[i] <= x):
                K[x] = max(K[x], K[x - w[i]] + v[i])
    return K[W]
```

Test Results

```
----------------------------------------------
weights: [5, 4, 2]
values: [10, 4, 3]
W: 9 and n: 3
Total value that can fit the knapsack: 16
----------------------------------------------
```

Complexity Analysis

In this algorithm we check all of them(0 to n) for all weights from 0 to W.

Therefore, our time complexity is;

T(n) Ɛ O(Wn)