# Gebze Technical University

# CSE 321 Homework 5

# Questions Report

Nevzat Seferoglu

171044024

## *Homework Topic:

In this homework, the concept that is requested to implement is all related to **dynamic programming**. I will explain each question and algorithm one by one. But before that, I want to make some brief introduction to dynamic programming. Because all solutions have in common point that is part of dynamic programming approach.

## *Dynamic Programming:

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure (array, map, etc.). Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, to facilitate its lookup. So, the next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time.

***Hint:** There are other solutions which is applied to dynamic programming problem. Those solutions may be better according to space complexity. But in dynamic programming, we actually do not care about the space complexity because of time efficiency.

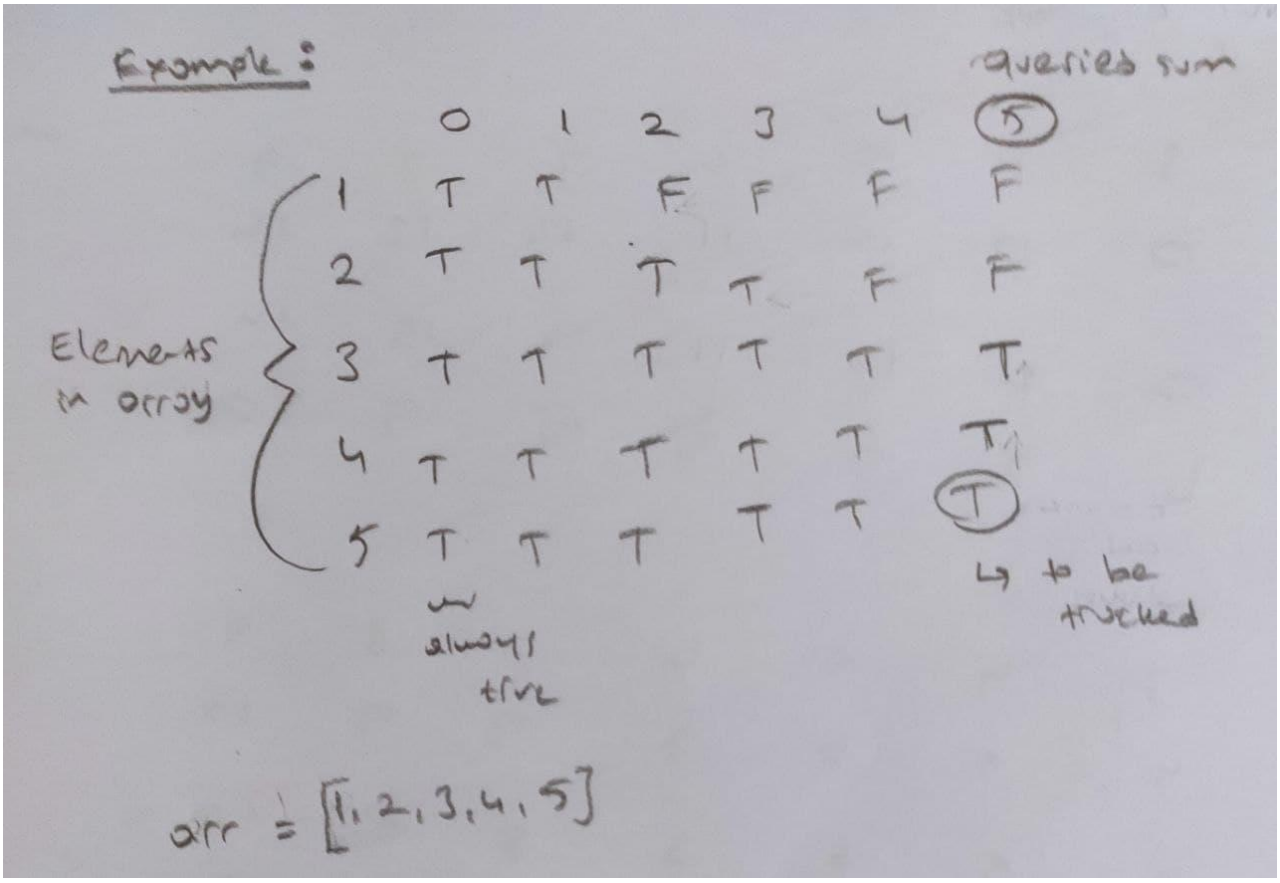
### *Memoization:

In computing, **memoization** or **memoisation** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

## Question – 1:

In this question, that was the classical subset sum problem. But the tricky part was negative numbers. Negative numbers, which exist in the list, are broking the rule of filling table of dynamic programming. I could not handle the negative numbers in the subset. I designed an algorithm which is only valid for array which has only positive numbers. In dynamic approach there is table that we record the value while scanning inputs and filling the outputs according to inputs. Indeed, filling is based on **excluding** and **including**.

For each element we compare the element with sum 'S'. Beginning with Nth element, if it is greater than 'S', we have only one choice i.e. exclude the Nth element. If the Nth element is less than or equal to 'S', we have two choices — include element or exclude it. If we include the Nth element, then we must reduce its value from sum 'S' and proceed with remaining N-1 elements. If we exclude the element, then the sum remains as 'S' and we can proceed with remaining N-1 elements. True and false can be used to check whether it is included or excluded while moving throughout bottom to top.



## Question – 2:

There is another dynamic approach in question two. The misunderstand of this question is that it looks like greedy approach, but it is not. By according to dynamic approach, we can create a way down throughout triangle, while keeping record of the smallest sum path for each element. If we are in first row or last row, the node has only one parent.

Therefore, we simply add parent's smallest path sum to the node's one. If we are in the middle, we add the smallest of the two parents' paths. When we get to the last row, we will have the minimum path sum for each number in the last row. We simply take the smallest of these minimums to find the minimum of the entire triangle. After that we simply print the result to display.

Note: The result is upper left corner of the table. This table for just demonstration for the example that is demonstrated on the assignment pdf file.

| Result: 14 | 0 | 0 | 0 |
|---|---|---|---|
| 12 | 14 | 0 | 0 |
| 7 | 10 | 13 | 0 |
| 8 | 6 | 9 | 6 |

## Question – 3:

This is different from the classical Knapsack problem, In this we are permitted to use an unlimited number of instances of an element. We will need a two-d array of size weight of W plus 1 and the number of elements plus 1. Next, we will move to initialization. The first row and the first column are initialized by 0. Next, we will be filling the remaining cell values in our array. Each item's weight is compared with the current weight. If the weight of the item is lesser than the weight the knapsack can hold, then we have 2 choices, **include** it in the current situation or **exclude** from the current situation. Let's talk about the table at given in assignment.

What is given:

```
# v1= 10, v2= 4, v3= 3;
value = [10, 4, 3]

# w1= 5, w2= 4, w3= 2;
weight = [5, 4, 2]

# capacity
W = 9

# size
size = len(value)
```

*Approach: