NOTES

Problem: Consider a game where a player can score 3 or 5 or 10 points at a time. Given a total score n, find the number of ways to reach the given score. The order of scoring does not matter.

Sample Input: n = 20
output: 4
Explanation: These are the following 4 ways to reach 20:(10, 10) (5, 5, 10) (5, 5, 5, 5) (3, 3, 3, 3, 3, 5)

Sample Input: n = 13
output: 2
Explanation: These are the following 2 ways to reach 13:(10, 3) (5, 5, 3)

Solution: let's assume that the array S contains the scores given and n be the total given score. For example, S = {3, 5, 10} and n can be 20, which means that we need to find the number of ways to reach the score 20 where a player can score either score 3, 5 or 10.

Optimal Sub-structure:
To count the total number of solutions, we can divide all set solutions into two sets.
1) Solutions that do not contain the i^{th} index score (or S[i]).
2) Solutions that contain at least one i^{th} index score.

Solution using dynamic programming

```
mp = {}

def count_dp(n, scores, end_index):
	#make a key
	key = str(n) + ":" + str(end_index);

	#check if key is there
	if(key in mp):
		return mp[key]

	val = 0;
	if n == 0:
		return 1
	elif n < 0 or end_index < 0:
		return 0
	elif scores[end_index] > n:
		val = count_dp(n, scores, end_index-1)
	else:
		val = count_dp(n - scores[end_index], scores, end_index) + count_dp(n, scores, end_index -
1);
	mp[key] = val;
	return val


scores = [3, 5, 10]
n = 13
print("Number of ways to score",n,"are",count_dp(n, scores, len(scores) - 1))
```

0-1 Knapsack Problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Sample input

val[] = {60, 100, 120}
weight[] = {10, 20, 30}
W = 50

Output: 220

Explanation:
For weight 10 + 20 values is 60 + 100 = 160
For weight 10 + 30 values is 60 + 120 = 180
For weight 30 + 20 values is 120 + 100 = 220

```
#Returns the maximum value that can be stored by the bag
def knapSack(W, wt, val, n):
    # initial conditions
    if n == 0 or W == 0 :
        return 0
    # If weight is higher than capacity then it is not included
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    # return either nth item being included or not
    else:
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),knapSack(W, wt, val, n-1))

# To test above function
val = [100, 120, 60]
wt = [20, 30, 10]
W = 50
n = len(val)
print (knapSack(W, wt, val, n))
```

Complexity: $O(2^n)$

0-1 Knapsack Problem using Dynamic programming

```
def knapSack(val, wt, W):
    n=len(val)
    table = [[0 for x in range(W + 1)] for x in range(n + 1)]

    for i in range(n + 1):
        for j in range(W + 1):
            if i == 0 or j == 0:
                table[i][j] = 0
            elif wt[i-1] <= j:
                table[i][j] = max(val[i-1]  + table[i-1][j-wt[i-1]],  table[i-1][j])
            else:
                table[i][j] = table[i-1][j]
    return table[n][W]

value = [100, 120, 60]
weight = [20, 30, 10]
capacity = 50
print(knapSack(value, weight, capacity))
```