Dynamic Programming

❖ 0-1 Knapsack Problem

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
Output: *3*

Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
Output: *0*

To solve the problem follow the below idea:

Since subproblems are evaluated again, this problem has an Overlapping Sub-problems property. So the 0/1 Knapsack problem has both properties of a dynamic programming problem.

Illustration:

Below is the illustration of the above approach:

Let, **weight[] = {1, 2, 3}, profit[] = {10, 15, 40}, Capacity = 6**

- If no element is filled, then the possible profit is 0.

weight⇢ item⇣/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

- **For filling the first item in the bag:** If we follow the above mentioned procedure, the table will look like the following.

weight⇢ item⇣/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2							
3							

- **For filling the second item:**
  When jthWeight = 2, then maximum possible profit is max ($\overset{V[0]}{10}$, DP[1][2-2] + $\overset{V[1]}{15}$) = max(10, 15) = 15.
  When jthWeight = 3, then maximum possible profit is max(2 not put, 2 is put into bag) = max(DP[1][3], $\underset{V[1]}{15}$+DP[1][3-2]) = max(10, 25) = 25.

weight⇢ item↓/	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- **For filling the third item:**

  When jthWeight = 3, the maximum possible profit is max(DP[2][3], $\overset{v[2]}{40}$+DP[2][3-3]) = max(25, 40) = 40.

  When jthWeight = 4, the maximum possible profit is max(DP[2][4], 40+DP[2][4-3]) = max(25, 50) = 50.

  When jthWeight = 5, the maximum possible profit is max(DP[2][5], 40+DP[2][5-3]) = max(25, 55) = 55.

  When jthWeight = 6, the maximum possible profit is max(DP[2][6], 40+DP[2][6-3]) = max(25, 65) = 65.

weight⇢ item↓/	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

C Implementation

```
#include <stdio.h>
#include <string.h>
int findMax(int n1, int n2){
  if(n1>n2) {
    return n1;
  } else {
    return n2;
  }
}

int knapsack(int W, int wt[], int val[], int n){
  int K[n+1][W+1];
  for(int i = 0; i<=n; i++) {
    for(int w = 0; w<=W; w++) {
      if(i == 0 || w == 0) {
        K[i][w] = 0;
      } else if(wt[i-1] <= w) {
        K[i][w] = findMax(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
      } else {
        K[i][w] = K[i-1][w];
      }
    }
  }
  return K[n][W];
}
int main(){
  int val[5] = {70, 20, 50};
  int wt[5] = {11, 12, 13};
  int W = 30;
  int len = sizeof val / sizeof val[0];
  printf("Maximum Profit achieved with this knapsack: %d", knapsack(W, wt, val, len));
}
```

Longest Common Subsequence (LCS)

Given two strings text1 and text2, return *the length of their longest* ***common subsequence****.* If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

Algorithm of Longest Common Subsequence

```
1.  Suppose X and Y are the two given sequences
2.  Initialize a table of LCS having a dimension of X.length * Y.length
3.  XX.label = X
4.  YY.label = Y
5.  LCS[0][] = 0
6.  LCS[][0] = 0
7.  Loop starts from the LCS[1][1]
8.  Now we will compare X[i] and Y[j]
9.     if X[i] is equal to Y[j] then
10.      LCS[i][j] = 1 + LCS[i-1][j-1]
11.      Point an arrow LCS[i][j]
12. Else
13.    LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
```