Longest Common Subsequence

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

But before we understand the problem, let us understand what the term subsequence is

Let us consider a sequence S = <s_1, s_2, s_3, s_4, ,s_n>. And another sequence Z = <z_1, z_2, z_3, ,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements. In simple words, a subsequence consists of consecutive elements that make up a small part in a sequence.

Common Subsequence

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

Naive Method

Let ***X*** be a sequence of length m and ***Y*** a sequence of length n. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***, and return the longest common subsequence found.

There are $\mathbf{2^m}$ subsequence's of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes O(n) time. Thus, the naive algorithm would take $O(n2^m)$ time.

Longest Common Subsequence Algorithm

Let X=<x_1,x_2,x_3....,x_m> and Y=<y_1,y_2,y_3....,y_m> be the sequences. To compute the length of an element the following algorithm is used.

Step 1 − Construct an empty adjacency table with the size, n m, where n = size of sequence **X** and m = size of sequence **Y**. The rows in the table represent the elements in sequence X and columns represent the elements in sequence Y.

Step 2 − The zeroth rows and columns must be filled with zeroes. And the remaining values are filled in based on different cases, by maintaining a counter value.

- **Case 1** − If the counter encounters common element in both X and Y sequences, increment the counter by 1.
- **Case 2** − If the counter does not encounter common elements in X and Y sequences at T[i, j], find the maximum value between T[i-1, j] and T[i, j-1] to fill it in T[i, j].

Step 3 − Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.

Step 4 − The longest common subsequence obtained by noting the elements in the traced path.

Example

In this example, we have two strings ***X=BACDB*** and ***Y=BDCB*** to find the longest common subsequence.

Following the algorithm, we need to calculate two tables 1 and 2.

Given n = length of X, m = length of Y

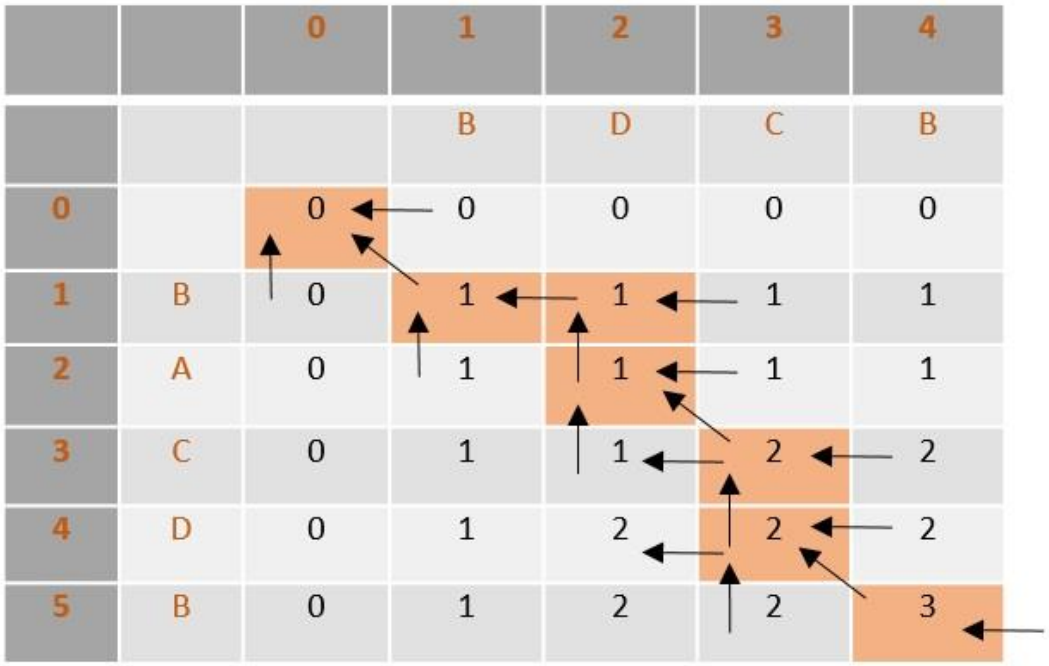
X = BDCB, Y = BACDB

Constructing the LCS Tables

In the table below, the zeroth rows and columns are filled with zeroes. Remaining values are filled by incrementing and choosing the maximum values according to the algorithm.

		0	1	2	3	4
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
5	B	0	1	2	2	3

Once the values are filled, the path is traced back from the last value in the table at T[4, 5].



From the traced path, the longest common subsequence is found by choosing the values where the counter is first incremented.

In this example, the final count is 3 so the counter is incremented at 3 places, i.e., B, C, B. Therefore, the longest common subsequence of sequences X and Y is BCB.

0-1 Knapsack Problem

Problem Statement − A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of i^{th} item is wi and the profit of selecting this item is ***pi***. What items should the thief take?

Let i be the highest-numbered item in an optimal solution **S** for **W** dollars. Then **S = S {i}** is an optimal solution for **W** w_i dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v1, v2, , vn> and w = <w1, w2, , wn>**

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If ***c[i, w] = c[i-1, w]***, then item i is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item i is part of the solution, and we continue tracing with **c [i-1, w-W]**.

Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7

Solution

Using the greedy approach of 0-1 knapsack, the weight thats stored in the knapsack would be A+B = 4 with the maximum profit 2 + 4 = 6. But, that solution would not be the optimal solution.

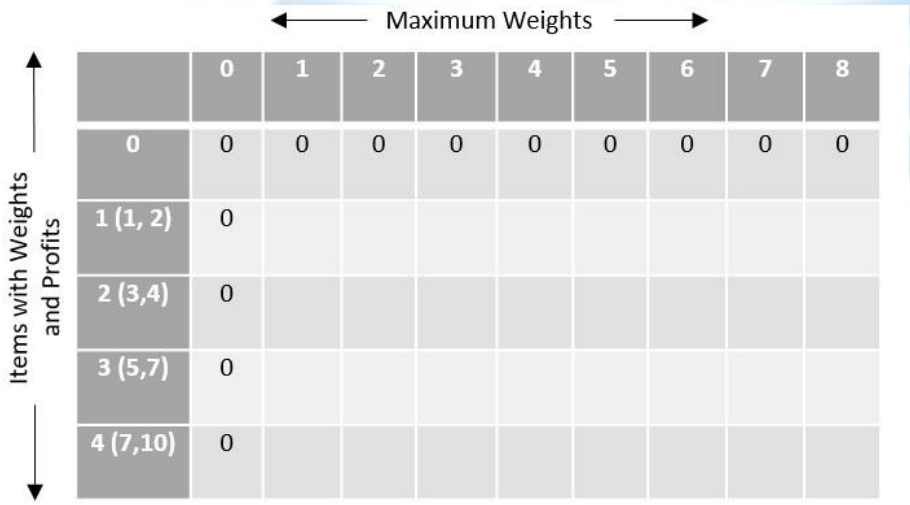
Therefore, dynamic programming must be adopted to solve 0-1 knapsack problems.

Step 1

Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.

Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row)

So we add zeroes to the 0^{th} row and 0^{th} column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.



The remaining values are filled with the maximum profit achievable with respect to the items and weight per column that can be stored in the knapsack.

The formula to store the profit values is −

$$c[i,w] = max\{c[i-1, w-w[i]] + P[i]\}$$

Maximum Weights →

Items with Weights and Profits ↓

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its {1, 7}.

Maximum Weights →

Items with Weights and Profits ↓

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	(12)

The optimal solution is {1, 7} with the maximum profit is 12.