DYNAMIC PROGRAMMING

- WHAT IS DYNAMIC PROGRAMMING?
- FIBONACCI SEQUENCE
- HOW DYNAMIC PROGRAMMING WORKS
- RECURSION VS DYNAMIC PROGRAMMING
- GREEDY ALGORITHMS VS DYNAMIC PROGRAMMING
- LONGEST COMMON SUBSEQUENCE
- THE KNAPSACK PROBLEM
- PRACTICE: LONGEST COMMON SUBSEQUENCE

Author: prof. Yevhenii Borodavka

WHAT IS DYNAMIC PROGRAMMING?

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems with overlapping subproblems and optimal substructure properties.

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. Thus, the efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

FIBONACCI SEQUENCE

Let's find the Fibonacci sequence up to 5th term. A Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding ones. For example, 0,1,1,2,3. Here, each number is the sum of the two preceding numbers.

Algorithm

Let **n** be the number of terms.

1. If **n** <= 1, return 1.
2. Else, return the sum of two preceding numbers.

FIBONACCI SEQUENCE

We are calculating the Fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is a sum of 0 (from step 1) and 1 (from step 2), 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. 1 + 1 = 2.
5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. 2 + 1 = 3.

FIBONACCI SEQUENCE

Hence, we have the sequence 0,1,1,2,3. Here, we have used the results of the previous steps as shown below. This is called a dynamic programming approach.

F(0) = 0

F(1) = 1

F(2) = F(1) + F(0)

F(3) = F(2) + F(1)

F(4) = F(3) + F(2)

HOW DYNAMIC PROGRAMMING WORKS

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

```
var m = map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] = fib(n − 1) + fib(n − 2)
    return m[n]
```

HOW DYNAMIC PROGRAMMING WORKS

Dynamic programming by memoization is a top-down approach to dynamic programming. By reversing the direction in which the algorithm works i.e. by starting from the base case and working towards the solution, we can also implement dynamic programming in a bottom-up manner (tabulation).

```
function fib(n)
    if n = 0
        return 0
    else
        var prevFib = 0, currFib = 1
        repeat n - 1 times
            var newFib = prevFib + currFib
            prevFib = currFib
            currFib  = newFib
    return currentFib
```

RECURSION VS DYNAMIC PROGRAMMING

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the Fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

RECURSION VS DYNAMIC PROGRAMMING

The recursive approach

```
int fib(int n)
{
    if (n <= 1)
      return n;
    return fib(n - 1) +
           fib(n - 2);
}
```

Time complexity **O**(2^n)
Space Complexity **O**(1)

The DP with tabulation

```
int fib(int n)
{
    int mem[n];
    mem[0] = 0;
    mem[1] = 1;
    for(int i=2; i<n; i++)
    {
        mem[i] = mem[i-2] +
                 mem[i-1];
    }
    return mem[n];
}
```

Time complexity **O**(n)
Space Complexity **O**(1)

GREEDY ALGORITHMS VS DYNAMIC PROGRAMMING

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.

However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

LONGEST COMMON SUBSEQUENCE

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

LONGEST COMMON SUBSEQUENCE

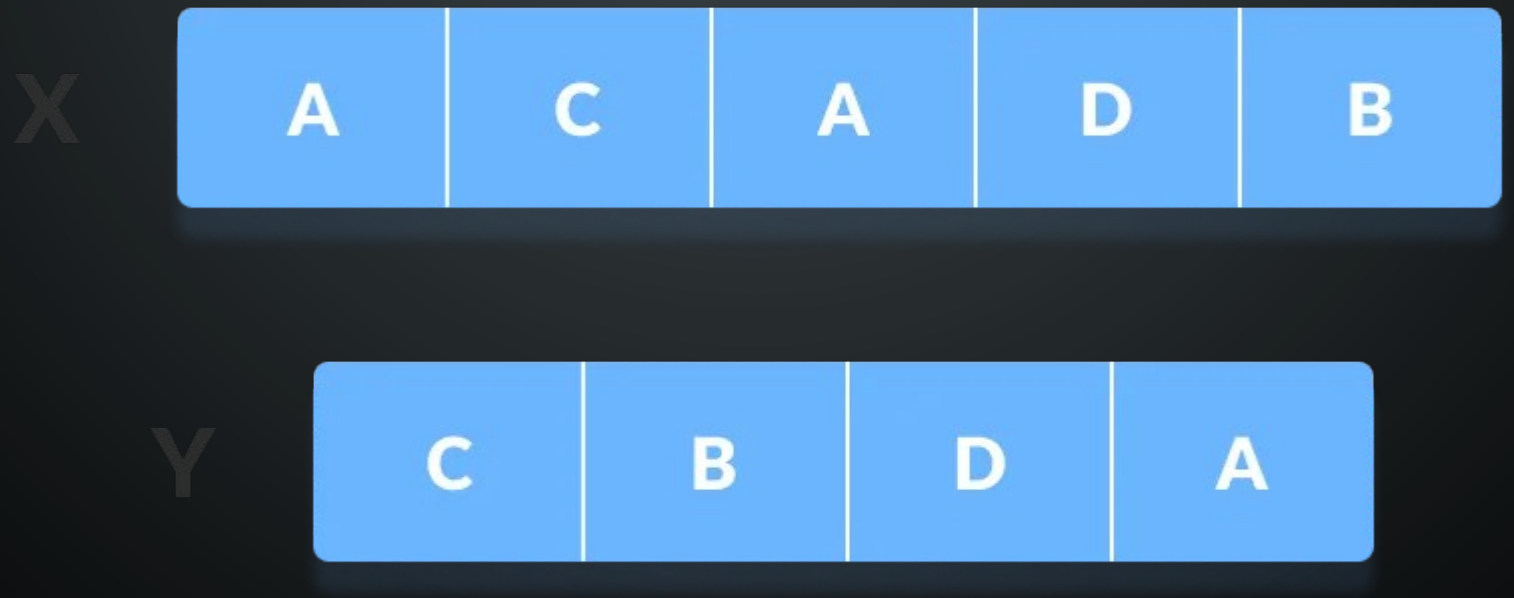
If S1 = {B, C, D, A, A, C, D} then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (i.e. not strictly increasing sequence).

If S1 = {B, C, D, A, A, C, D} and S2 = {A, C, D, B, A, C} then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, ...

Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

LONGEST COMMON SUBSEQUENCE

Let us take two sequences:



LONGEST COMMON SUBSEQUENCE

The following steps are followed for finding the longest common subsequence.

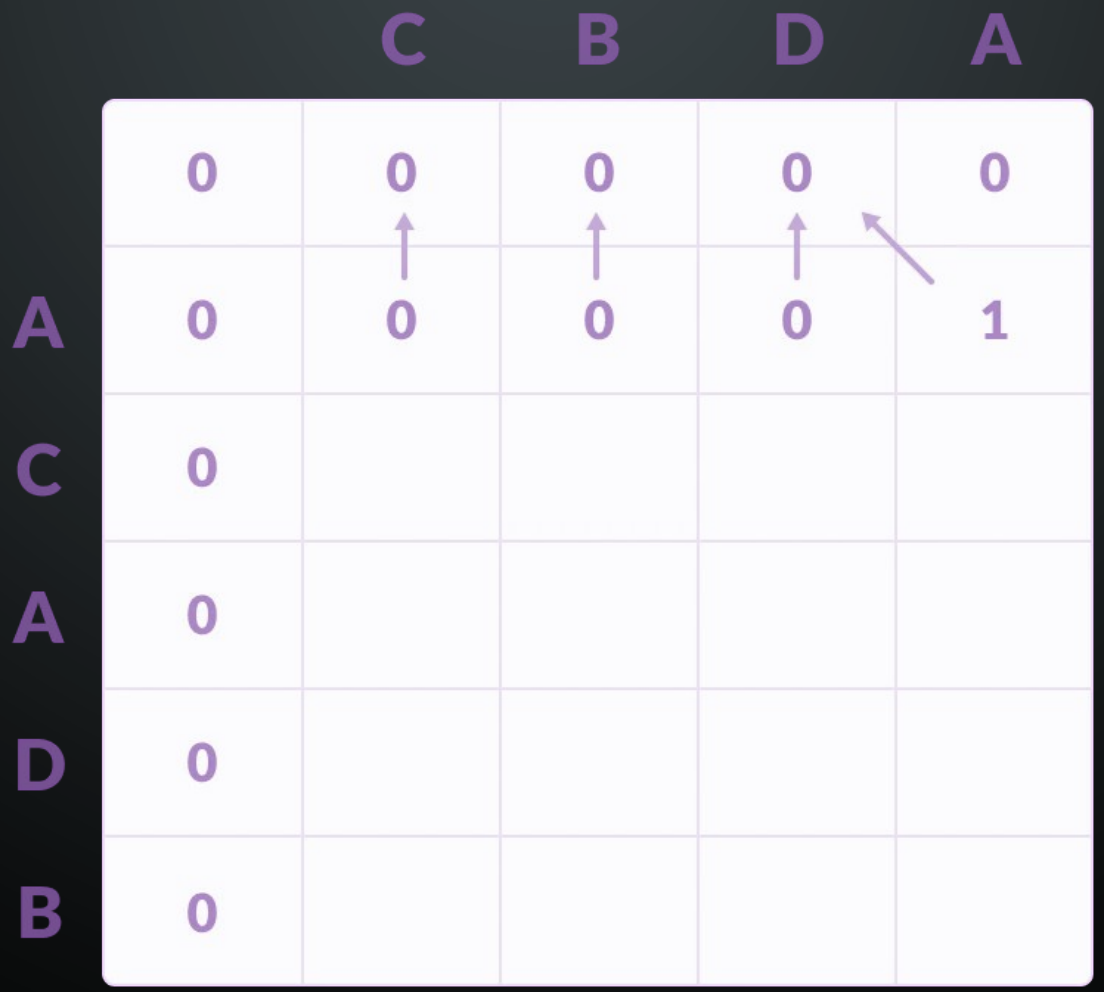
1. Create a table of dimension (n+1)*(m+1) where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

LONGEST COMMON SUBSEQUENCE

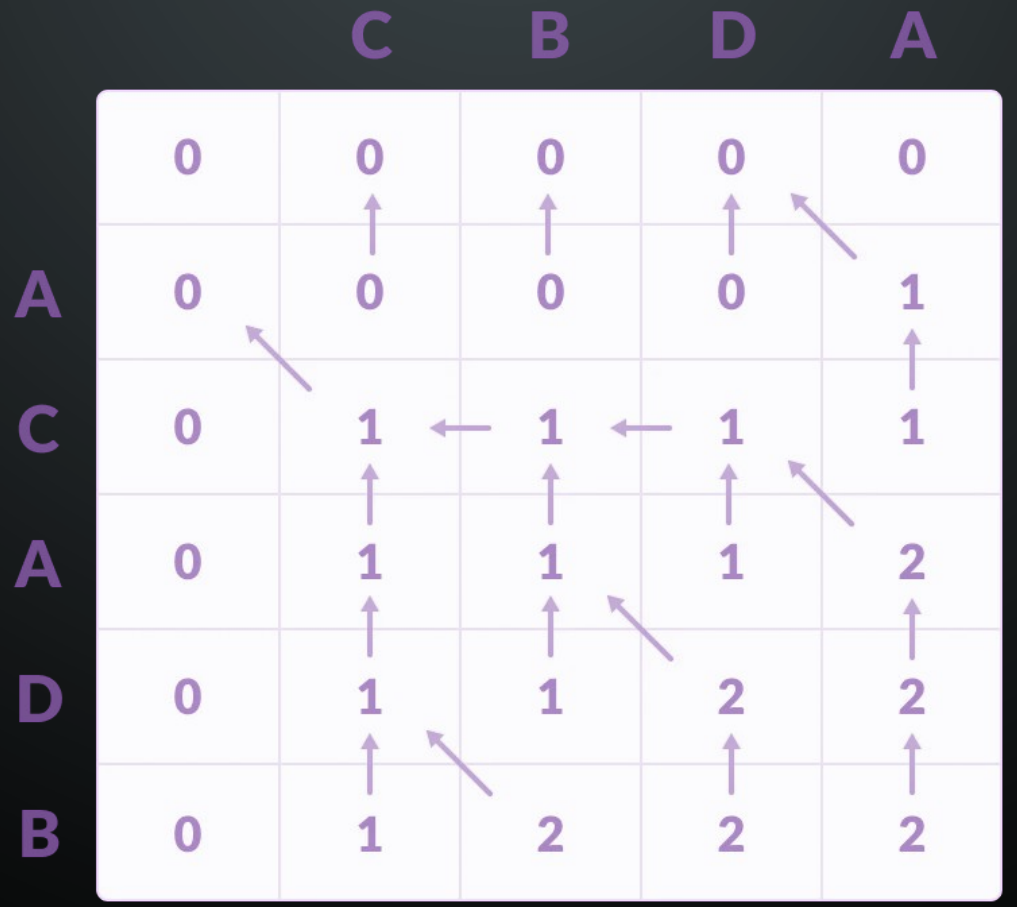
2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

LONGEST COMMON SUBSEQUENCE



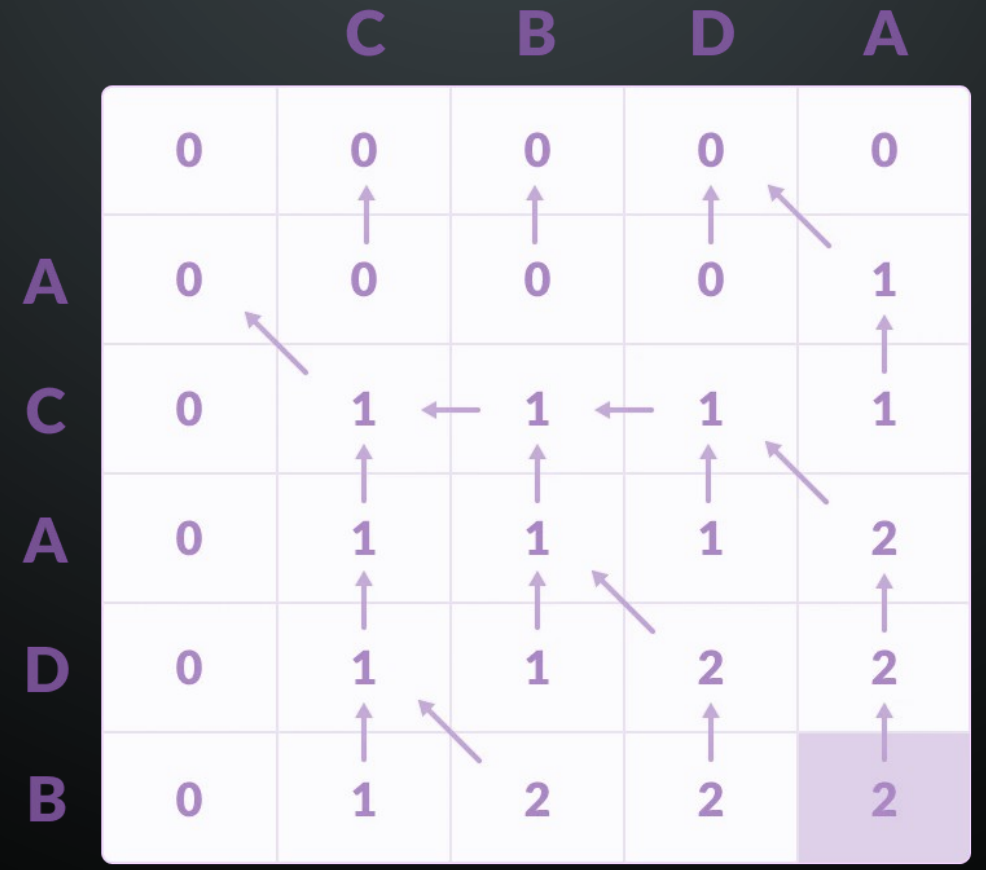
LONGEST COMMON SUBSEQUENCE

5. Step 2 is repeated until the table is filled



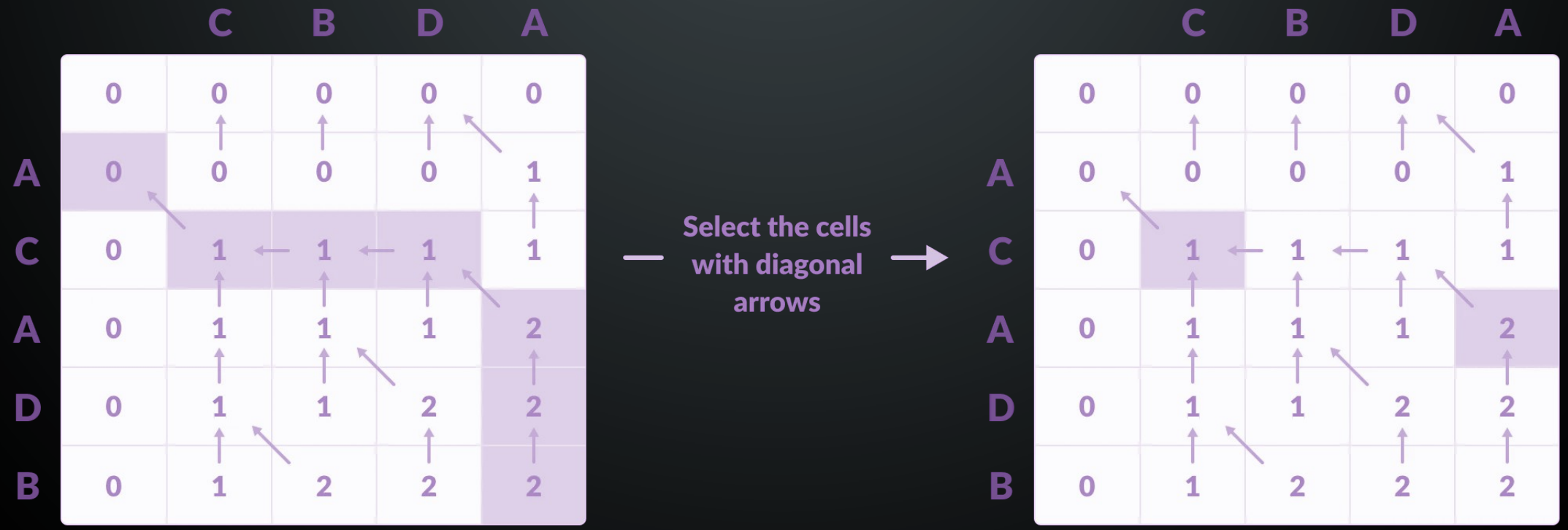
LONGEST COMMON SUBSEQUENCE

6. The value in the last row and the last column is the length of the longest common subsequence.



LONGEST COMMON SUBSEQUENCE

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow.



LONGEST COMMON SUBSEQUENCE

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of **X** and the elements of **Y** are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (i.e. **O**(m*n)). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

THE 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.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

THE KNAPSACK PROBLEM

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than **W**. From all such subsets, pick the subset with maximum profit.

Optimal Substructure: To consider all subsets of items, there can be two cases for every item.

- Case 1: The item is included in the optimal subset.
- Case 2: The item is not included in the optimal set.

THE KNAPSACK PROBLEM

Follow the below steps to solve the problem:

The maximum value obtained from **N** items is the max of the following two values.

- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (**W** — weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and **W** weight.
- If the weight of the Nth item is greater than **W**, then the Nth item cannot be included and Case 2 is the only possibility.

THE KNAPSACK PROBLEM

Naïve recursive implementation

```
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n -
1),
                   knapSack(W, wt, val, n - 1));
}
```

THE KNAPSACK PROBLEM

Time Complexity: **O**(2^N). Auxiliary Space: **O**(N).

Note: It should be noted that the above function using recursion computes the same subproblems again and again.

As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem.

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the Knapsack problem has both properties of a dynamic programming problem. Like other typical DP problems, re-computation of the same subproblems can be avoided by constructing a

THE KNAPSACK PROBLEM

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							

THE KNAPSACK PROBLEM

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							

THE KNAPSACK PROBLEM

For filling the second item:

For j>=2, then maximum profit is max (DP[1][j], w[2]+DP[1][j-2])

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							

THE KNAPSACK PROBLEM

For filling the third item:

For j>=3, then maximum profit is max (DP[2][j], w[3]+DP[2][j-3])

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

THE KNAPSACK PROBLEM

Time Complexity: **O**(N*W). where **N** is the number of elements and **W** is capacity.

Auxiliary Space: **O**(N*W). The use of a 2-D array of size N*W.

Space optimized Approach for the Knapsack Problem using Dynamic Programming:

For calculating the current row of the DP[] array we require only previous row, but if we start traversing the rows from right to left then it can be done with a single row only. In this case we use 1-D array of size W.

PRACTICE: LONGEST COMMON SUBSEQUENCE

Problem. You have two strings with lengths from 1 to 1000 symbols. Find the longest common subsequence of these two strings.

Task. Create a program using C/C++/Python to solve this problem.

Input. Two strings are divided by space.

Output. The length of the longest common subsequence.

PRACTICE: LONGEST COMMON SUBSEQUENCE

Code example:

```
#include <iostream>
#define MAX 1000

char S[MAX];
int A[MAX][MAX];

int main()
{
    cin >> S;

    /*your code here*/

    cout << A[MAX-1][MAX-1];

    return 0;
}
```

THANK YOU!