Agenda

- What is Dynamic Programming?
- Top-down and Bottom-up dynamic programming
- Problem: Longest Common Subsequence
- Problem: 0-1 Knapsack
- Problem: Edit Distance (Only top-down)
- Problem: Coin Change - Minimum number of coins (Only top-down)

What is Dynamic Programming?

In simple words, it is a book-keeping technique that simply means the act of storing the results and using them for future purposes (to save time and resources).

It also gives us a life lesson - Make life less complex. There is no such thing as a big problem in life. Even if it appears big, it can be solved by breaking into smaller problems and then solving each optimally.

Let's consider a conversion between A and B to understand the true essence of dynamic programming:
A *writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper*
A : "What's that equal to?"
B : *counting* "Eight!"
A *writes down another "1+" on the left*
A : "What about that?"
B : *quickly* "Nine!"
A : "How'd you know it was nine so fast?"
B : "You just added one more"
A : "So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say 'remembering stuff to save time later'"

The idea is very simple. If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. shortly ‘Remember your Past’. If the given problem can be broken up into smaller subproblems and these smaller subproblems are in turn divided into still-smaller ones, and in this process, if you observe some **overlapping subproblems**, then it is a big hint for DP. Also, the **optimal solutions** to the subproblems contribute to the optimal solution of the given problem.

Top-down and Bottom-up dynamic programming

Top-down DP:
Memoization
It is recursive in nature and while computing the result of a problem, we **assume** that all the results of its subproblems have already been computed. That is, we make recursive calls to those subproblems, which further make recursive calls to their respective subproblems and so on..
For example, to compute fib(100), we will assume that we already have the value of fib(99) and fib(98) (by making the respective recursive calls) and will use those results to compute fib(100).

Bottom-up Dp:
Tabulation
This is a kind of a "table-filling" algorithm. We need to think of the **exact order** of computation of all of the subproblems.
For example, to compute fib(100), we will start by computing fib(0), fib(1), fib(2), fib(3), fib(4) …… fib(99) in the exact same order.

Memoization is generally easier to implement compared to tabulation but being recursive in nature, it is sometimes slower compared to the tabulation method.

Longest Common subsequence

Given two sequences, find the length of longest subsequence present in both of them. Both the strings are of uppercase.

Example 1:

```
Input:
A = 6, B = 6
str1 = ABCDGH
str2 = AEDFHR
Output: 3
Explanation: LCS for input Sequences
“ABCDGH” and “AEDFHR” is “ADH” of
length 3.
```

Example 2:

```
Input:
A = 3, B = 2
str1 = ABC
str2 = AC
Output: 2
Explanation: LCS of "ABC" and "AC" is
"AC" of length 2.
```

0-1 Knapsack

Given two integer arrays *values* and *weights* of size N each 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 such that sum of the weights of this subset is smaller than or equal to W.

NOTE: You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Input-1:
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50

Output-1:
220

Input-2:
values = [60, 100, 120]
weights = [10, 20, 50]
W = 50

Output-2:
160

Edit Distance

Given two strings **A** and **B**, find the minimum number of steps required to convert **A** to **B**. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Examples:

```
Input 1:
    A = "abad"
    B = "abac"

Output 1:
    1

Explanation 1:
    Operation 1: Replace d with c.

Input 2:
    A = "Anshuman"
    B = "Antihuman"

Output 2:
    2

Explanation 2:
    => Operation 1: Replace s with t.
    => Operation 2: Insert i.
```

Coin Change - Minimum number of coins

You are given an amount denoted by **value**. You are also given an array of coins. The **array** contains the **denominations** of the given coins. You need to find the **minimum number of coins** to make the change for **value** using the coins of given denominations. Also, keep in mind that you have **infinite supply** of the coins.

Example 1:

Input:
value = 5
numberOfCoins = 3
coins[] = {3,6,3}
Output: Not Possible
Explanation: We need to make the change for value = 5 The denominations are {3,6,3}
It is certain that we cannot make 5 using any of these coins.

Example 2:

Input:
value = 10
numberOfCoins = 4
coins[] = {2 5 3 6}
Output: 2
Explanation: We need to make the change for value = 10 The denominations are {2,5,3,6}
We can use two 5 coins to make 10. So minimum coins are 2.