Agenda

- What is Dynamic Programming?
- Top-down and Bottom-up dynamic programming
- Problem: Longest Common Subsequence
- Problem: 0-1 Knapsack
- Problem: nCr (just 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 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?"
A : "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 sub-problems 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, its 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

nCr

Find **^nC_r** for given **n** and **r**.

Example 1:

```
Input:
n = 3, r = 2
Output: 3
```

Example 2:

```
Input:
n = 4, r = 2
Output: 6
```

Your Task:
Complete the function **nCrModp()** which takes two integers **n** and **r** as input parameters and returns the **^nC_r mod** 10^9+7.