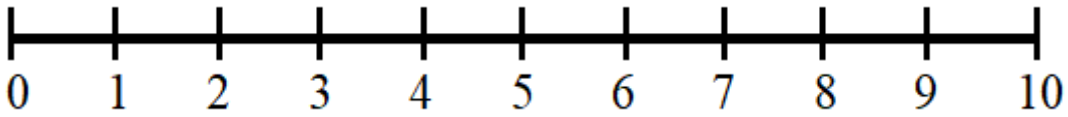
Dynamic Programming 2

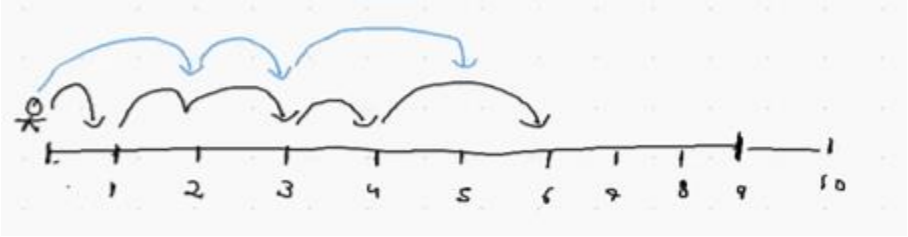
Topics:

- Problems on 1D & 2D

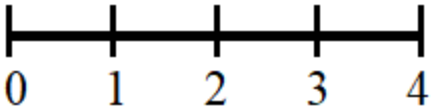
Q) Suppose you have a ladder (on x-axis) with steps from 0 – 10.



Now, you are 0 and you have two choices. You can take one step or you can take two steps.



If you have two choices, i.e., to take 1 step or 2 steps then find the number of ways by which you can reach the end of the ladder. Ex:



What are the number of ways you can reach the last step?

Tricks (to identify if we should use DP)

- Min / max is asked
- Recursion
- Count / no. of ways

Number of ways to reach 4 from 0.

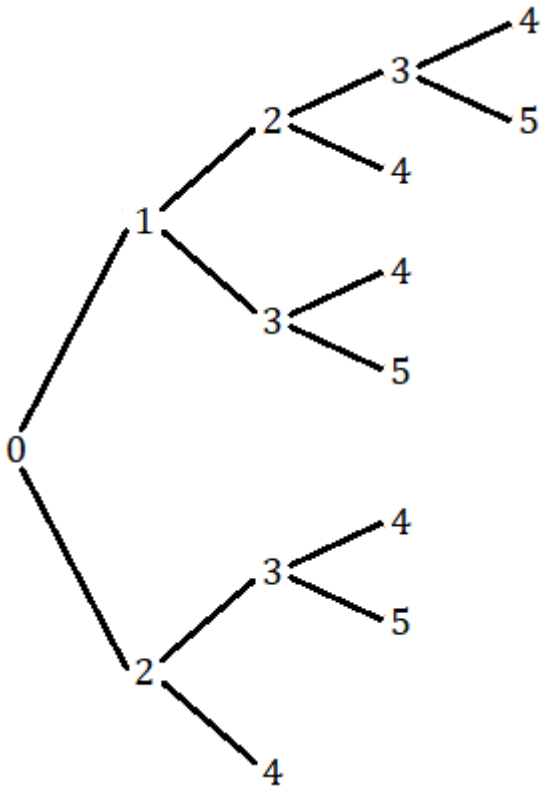
1st way ➔ 0 – 1 – 3 – 4

2nd way ➔ 0 – 1 – 2 – 3 – 4

3rd way ➔ 0 – 2 – 4

4th way ➔ 0 – 1 – 2 – 4

5th way ➔ 0 – 2 – 4



Since we can either take a jump of length 1 or 2, we have two choices at each step.

From 0,

If we take 1 jump of length 1, we will reach 1
if we take 2 jump of length 2, we will reach 2

From 0 – 1,

If we take jump of length 1, we will reach 2
If we take jump of length 2, we will reach 3

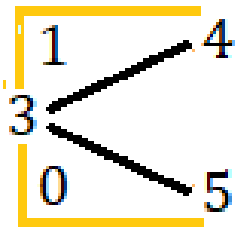
From 0 – 2,

If we take jump of length 1, we will reach 3
If we take jump of length 2, we will reach 4

From 0 – 2 – 3

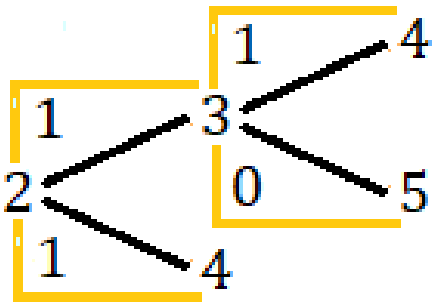
If we take jump of length 1, we will reach 4
If we take jump of length 2, we will reach 5. 5 is not a valid state, as there is no path possible and hence it would return 0

So, for every path that ends with 5, it would return 0 to its pervious place 4.



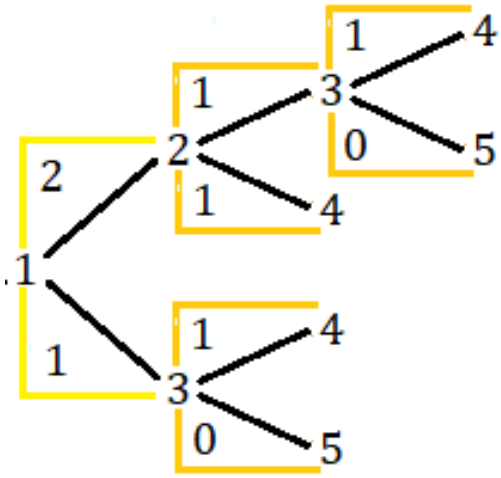
In the diagram, from 5 – 3, it would return 0 as 5 is not a valid state and from 4 it would return 1. So, the return value at 3 would be (left + right = 1 + 0) = **1**. So the value at 3 is 1.

At 2,



since the value at 3 is 1, we need to return 1 to 2 and from 4 as well, it would return 1. So the return value at 2 would be return value from 3 + return value from 4 = 1 + 1 = **2**

At 1,

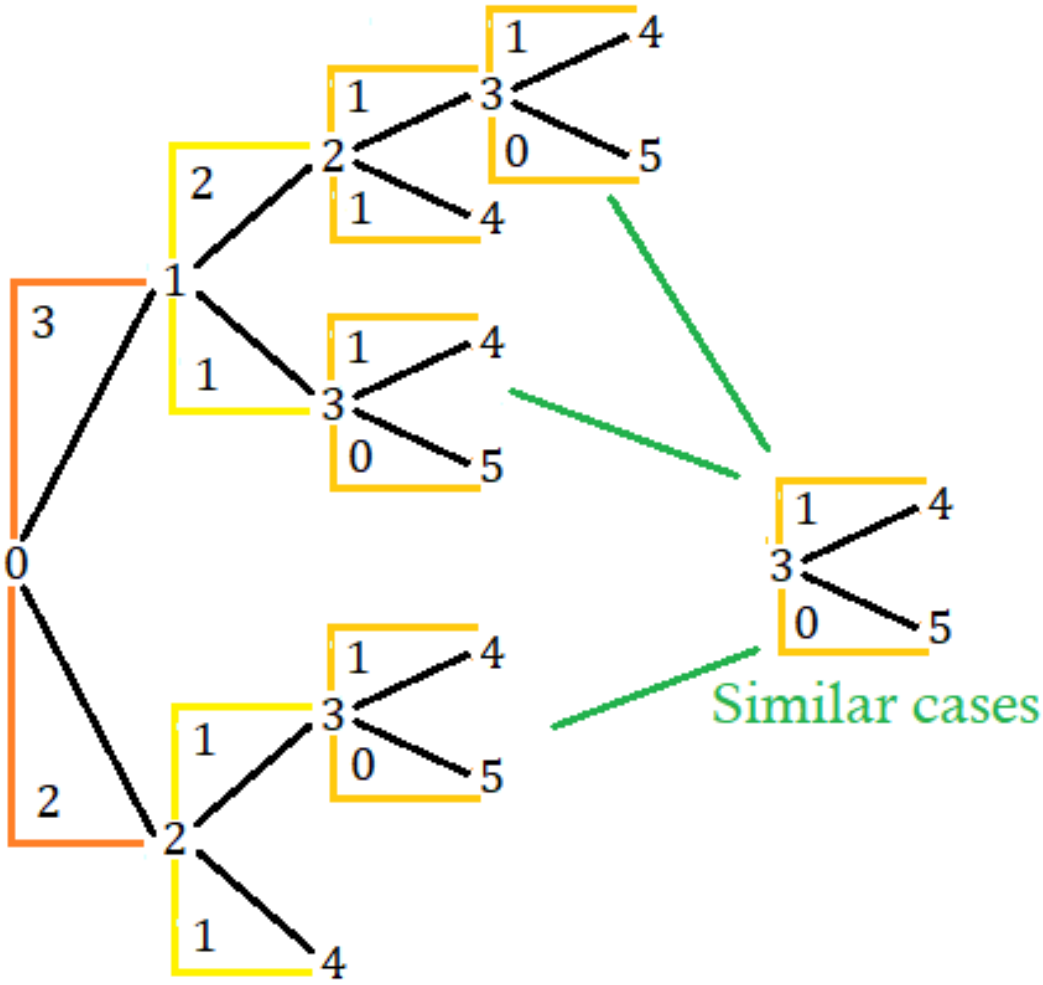


The value at 1 – 2 is 2, so it would return the value 2.

Now, the second choice of taking jump of length 2, will take us from 1 – 3. And from 3 with two more options, we would have two more paths, 1 – 3 – 4 and 1 – 3 – 5.

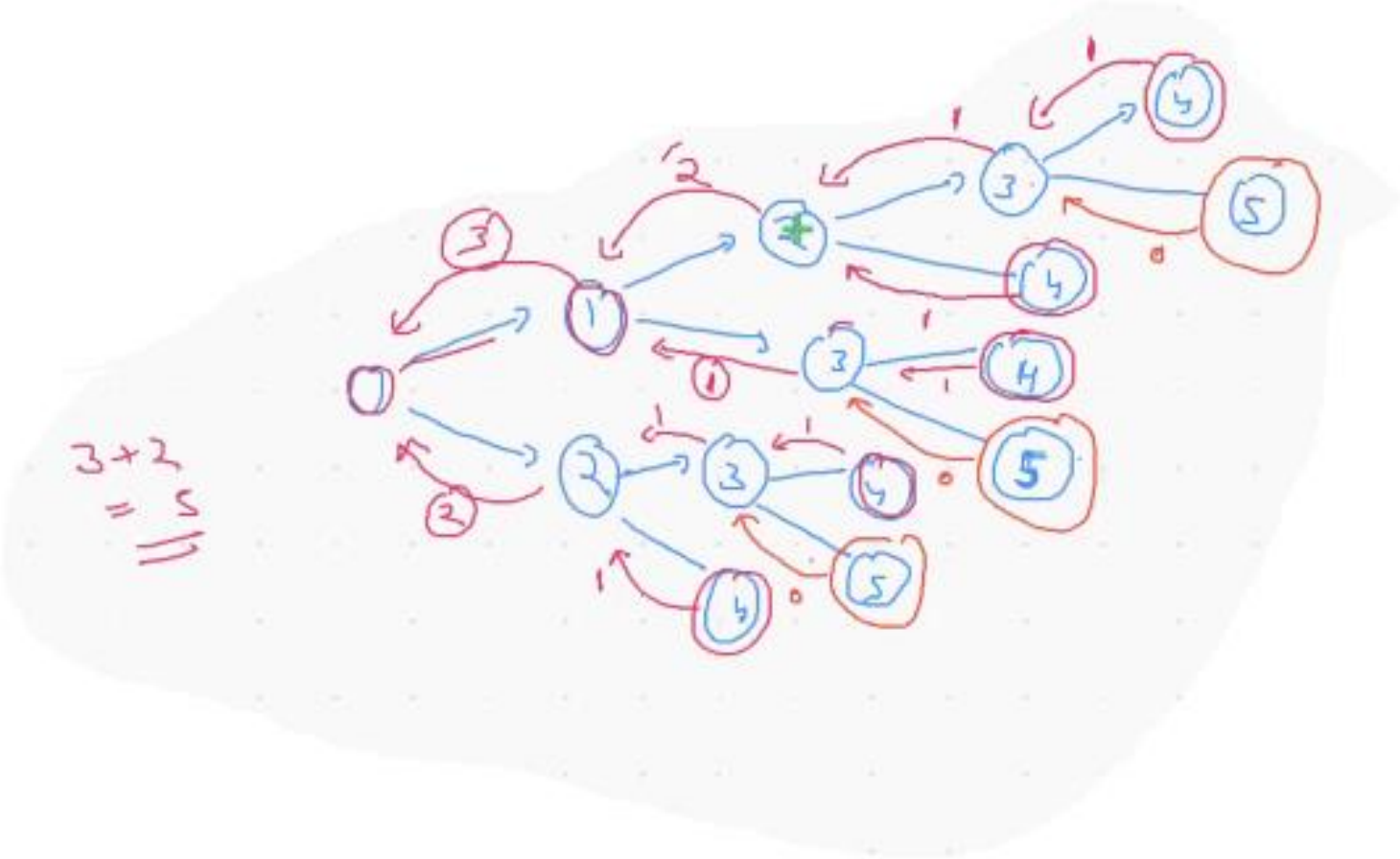
Since we have already established that 5 is not a valid state, hence it would return 0. This case is similar to the first case we have examined. So, based on that we can return the value at 3 as 1.

So, the value at 1 – 2 is 2 and the value at 1 – 3 is 1. Adding both the values, we would get the result, 2 + 1 = 3.



Since 3 – 4 & 3 – 5 are similar cases being repeated again, we know the value at every 3 would be 1. In recursion, we would find the value every time we come across such similar cases, but when using Dynamic Programming, the values are stored and this optimizes the code.

Hence value at 0 – 2 would be 2 because the value at 0 – 2 – 3 is 1 and the value for 0 – 2 – 4 is also 1. So adding the return value at left and return value at right will give us **2**.



CODE:

Recursive Solution

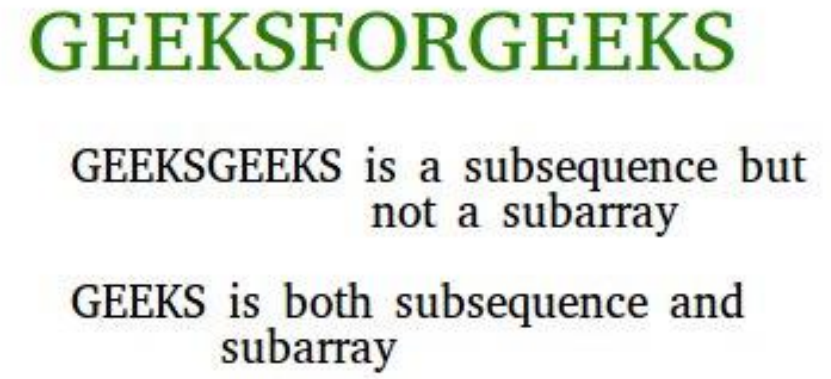
```
def solve(starting_position, n):
    if starting_position == n:
        return 1

    if starting_position > n:
        return 0

    left = solve(starting_position + 1, n)
    right = solve(starting_position + 2, n)
    return left + right

if __name__ == "__main__":
    n = 4
    solve(0, n)
```

DP Solution:

```
# Memoization rule - Whatever you take make an array of it.

def solve(starting_position, n):
    if starting_position == n:
        return 1
    if starting_position > n:
        return 0

    if dp[starting_position] != None:
        return dp[starting_position]

    left = solve(starting_position + 1, n)
    right = solve(starting_position + 2, n)

    dp[starting_position] = left + right
    return dp[starting_position]

if __name__ == "__main__":
    n = 4
    dp = [None for i in range[10005]]
    print(solve(0, n, dp))
```

Substring

A substring is a **contiguous** part of array. An array that is inside another array. For example, consider the array [1, 2, 3, 4], There are 10 non-empty sub-arrays. The substring is (1), (2), (3), (4), (1,2), (2,3), (3,4), (1,2,3), (2,3,4) and (1,2,3,4). In general, for an array/string of size n, there are **n*(n+1)/2** non-empty s/substrings.



Subsequence
A subsequence is a sequence that can be derived from another sequence by zero or more elements, without changing the order of the remaining elements.

For the same example, there are 15 sub-sequences. They **are (1), (2), (3), (4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4), (1,2,3), (1,2,4), (1,3,4), (2,3,4), (1,2,3,4).** More generally, we can say that for a sequence of size n, we can have (2^n**-1**) non-empty sub-sequences in total.

A string example to differentiate:

Consider strings “geeksforgeeks” and “gks”. “gks” is a subsequence of “geeksforgeeks” but not a substring. “geeks” is both a subsequence and substring. Every substring is a subsequence. More specifically, **Subsequence is a generalization of substring.**

A B C D G H = X

A E D F H R = Y

In X & Y we have A, D and H are common. So, the subsequence for both the strings would be A D H and the length of the longest subsequence would be 3.

X = A A R A M X Y

Y = B A M Y

In these two strings, A, M and Y are common. So, the subsequence for both the stings would be A M Y and the length of the longest subsequence would be 3.

Ex:

A S H I S H

R A B I S T

Explanation:

Let’s say we have two indices. Idx1 pointing at A and idx2 pointing at R. When will the length of the subsequence increase? If both the characters are same, then the length of subsequence is increased and will do recursion for further part by moving the index to the next element as well.

We have two choices in this we can either increase the idx1 or idx2.

Ex:

A P P L E

A R L E

Implementation:

Initially our idx1 is at A of X and idx2 is at A of Y. Since idx1 = idx2, both the index would be increased and hence idx1 = P and idx2 = R. since P & R are unequal, we have two choices.

The first choice would be we can keep the idx1 at P and increase the idx2 to L and the second choice would be we change the idx1 to 2nd P in (A P P L E) and keep the idx2 at the same character.

CODE:

```
def solve(str1, str2, idx1, idx2):

    # if anyone of the string is empty, return 0
    if idx1 >= len(str1) or idx2 >= len(str2):
        return 0

    if str1[idx1] == str2[idx2]:
        return 1 + solve(str1, str2, idx1 + 1, idx2 + 1)
    else:
        return max(solve(str1, str2, idx1 + 1, idx2), solve(str1, str
2, idx1, idx2 + 1))

if __name__ == "__main__":
    n = 4
    print(solve("ABCED", "BED", 0, 0))
```

```
# DP Approach

dp = [[None for _ in range(1000)] for _ in range(1000)]
def solve(str1, str2, idx1, idx2):

    if idx1 == len(str1) or idx2 == len(str2):
        return 0

    if dp[idx1][idx2] is not None:
        return dp[idx1][idx2]

    ans = 0
    if str1[idx1] == str2[idx2]:
        ans = 1 + solve(str1, str2, idx1 + 1, idx2 + 1)
    else:
        ans = max(solve(str1, str2, idx1 + 1, idx2), solve(str1, str2
, idx1, idx2 + 1))

    return ans

if __name__ == "__main__":
    n = 4
    print(solve("ABCED", "BED", 0, 0))
```

Resources:

For Dynamic Programming

1. https://www.geeksforgeeks.org/dynamic-programming/
2. https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/
3. https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/