PART 3

DP Type : LCS (Longest Common Subsequence)

```
def longestCommonSubsequence(text1, text2):
    n = len(text1)
    m = len(text2)
    def helper(index1, index2):
        if index1 > n-1 or index2 > m-1 :
            return 0
        if text1[index1] == text2[index2] :
            return 1 + (helper(index1+1, index2+1))
        return max(helper(index1, index2+1) ,
                   helper(index1+1, index2))
    return helper(0,0)
```

For the memoization approach, we just create an array of the following size:

```
[[-1] * m for _ in range(n)]:
```

and store the respective function call in the array.

Tabulation Method:

Tabulation Approach:

```
def longestCommonSubsequence(text1, text2):
    n, m = len(text1), len(text2)
    #creating the dp table with 0 included
    dp = [[0] * (m+1) for _ in range(n+1)]
    #filling the table
    for i in range(1, n+1):
        for j in range(1, m+1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j],
                               dp[i][j-1])

    return dp[n][m]
```

For space optimised tabulation approach we form two arrays:

```
prev = [0] * m+1
curr = [0] * m+1
```

and then replace dp[i-1] with prev and dp[i] with curr

Question : 72. Edit Distance

Input : word1 = "horse", word2 = "ros"

Output = 3

Explanation : We would do 3 operations to convert word1 to word2

The 3 allowed operation are :

INSERT, DELETE, REPLACE

We will use a approach similar to previous question but with some tweaks as there are three operation involved here.

For INSERT : Lets take a example

word 1 : horse , word 2 = "ros"

red represent the current pointer

Since both word are not same we do insert operation :

word 1 : 'r horse' word 2 = "ros"

we shift the pointer of word 2 as "r" is successfully matched

Similarly for the DELETE operation we shift the pointer of word1.

FOR Replace operation we shift the pointer of both words.

CODE :

```
def minDistance(word1, word2):
    n, m = len(n), len(m)
    # memoization table
    dp = [[-1] *m for _ in range(n)]]
    def helper(index1, index2):
        if index1>n-1 and index2 > m-1 :
            return 0
        if index 1 > n-1 :
            return m - index2
        if index2 > m-1
            return n - index 1
        if word1[index1] == word2[index2]:
            dp[index1][index2] = helper(index1+1
                                        index2+1)
            return dp[index1][index2]
        insert = helper(index1, index2+1)
        delete = helper(index1+1, index2)
        replace = helper(index1+1, index2+1)
        dp[index1][index2] = min(insert, delete,
                                 replace)
```

return dp[index1][index2]

#Tabulation approach

```
def minDistance(word1, word2):
    n, m = len(word1, word2)
    dp = [[0]*(m+1) for _ in range(n+1)]
    #filling the dp table for intial condition
    for i in range(n+1):
        dp[i][0] = i
    for j in range(m+1):
        dp[0][j] = j
    for i in range(1 n+1):
        for j in range(1, m+1):
            if word1[i-1] = word2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                insert = dp[i][j-1]
                delete = dp[i-1][j]
                replace = dp[i-1][j-1]
                dp[i][j] = min(insert, delete, replace)
    return dp[n][m]
```

QUESTION: Longest Increasing Subsequence

Input : [10, 9, 2, 5, 3, 7, 101, 18]

Output : 4

Explanation: [2, 3, 7, 101]

Let take an example to draw the recursion tree

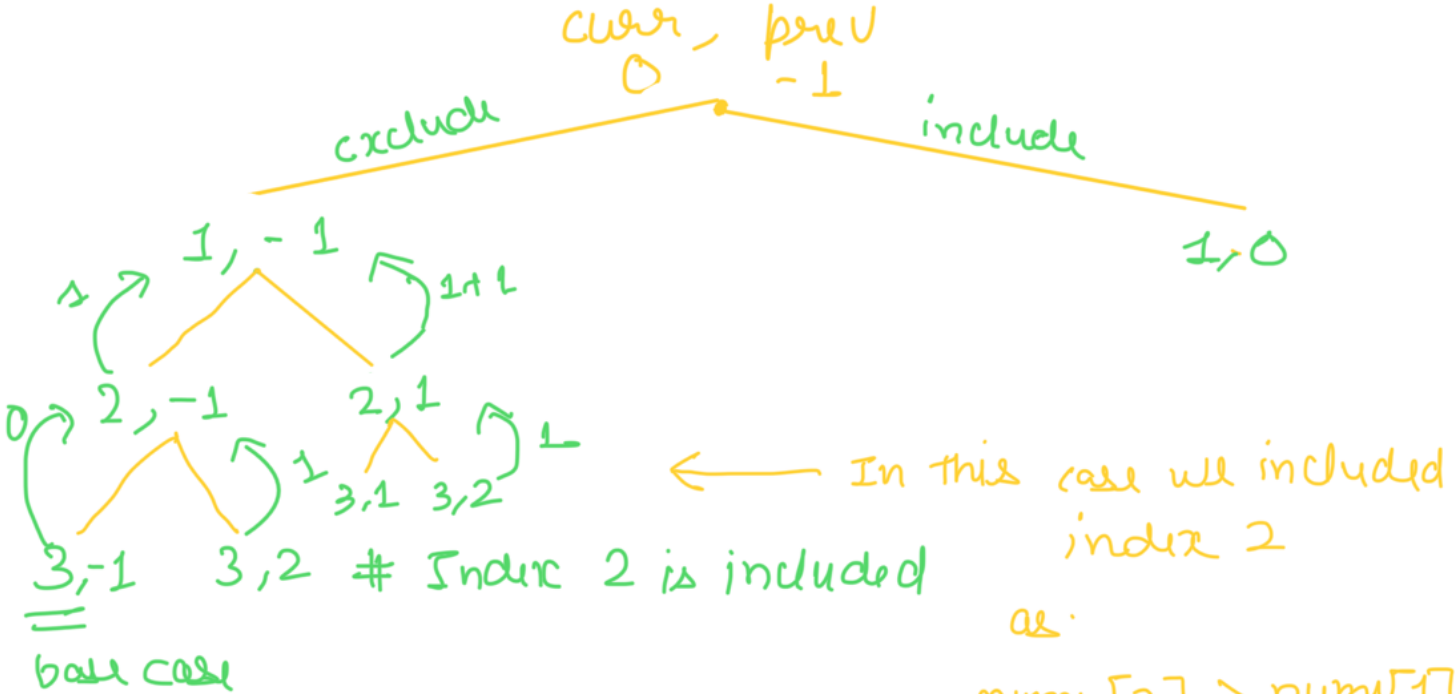
[1, 2, 3]

We will solve this question in a similar way in which we solved the "Subsets"

but with some conditions using curr and prev

We will start with curr = 0 and prev = -1

Prev = -1 represents that we haven't included any number.



CODE : Recursion + Memoization

```
def lengthofLIS(nums):
    n = len(nums):
    # Memoization dp table
    dp = [[-1]*n for _ in range(n)]
    def helper(curr_index, prev_index):
        if curr_index >n-1 :
            return 0
        # prev_index start from -1 so we add 1
        if dp[curr_index][prev_index+1] != -1:
            return dp[curr_index][prev_index+1]
        exclude = helper(curr_index+1, prev_index)
        include = 0
        if prev_index == -1 or nums[curr_index] >
                                num[prev_index]:
            include = helper(curr_index+1, curr_index)
        dp[curr_index][prev_index+1] = max(exclude,
                                           include)
        return dp[curr_index][prev_index+1]
    return helper(0,-1)
```

* Tabulation Approach

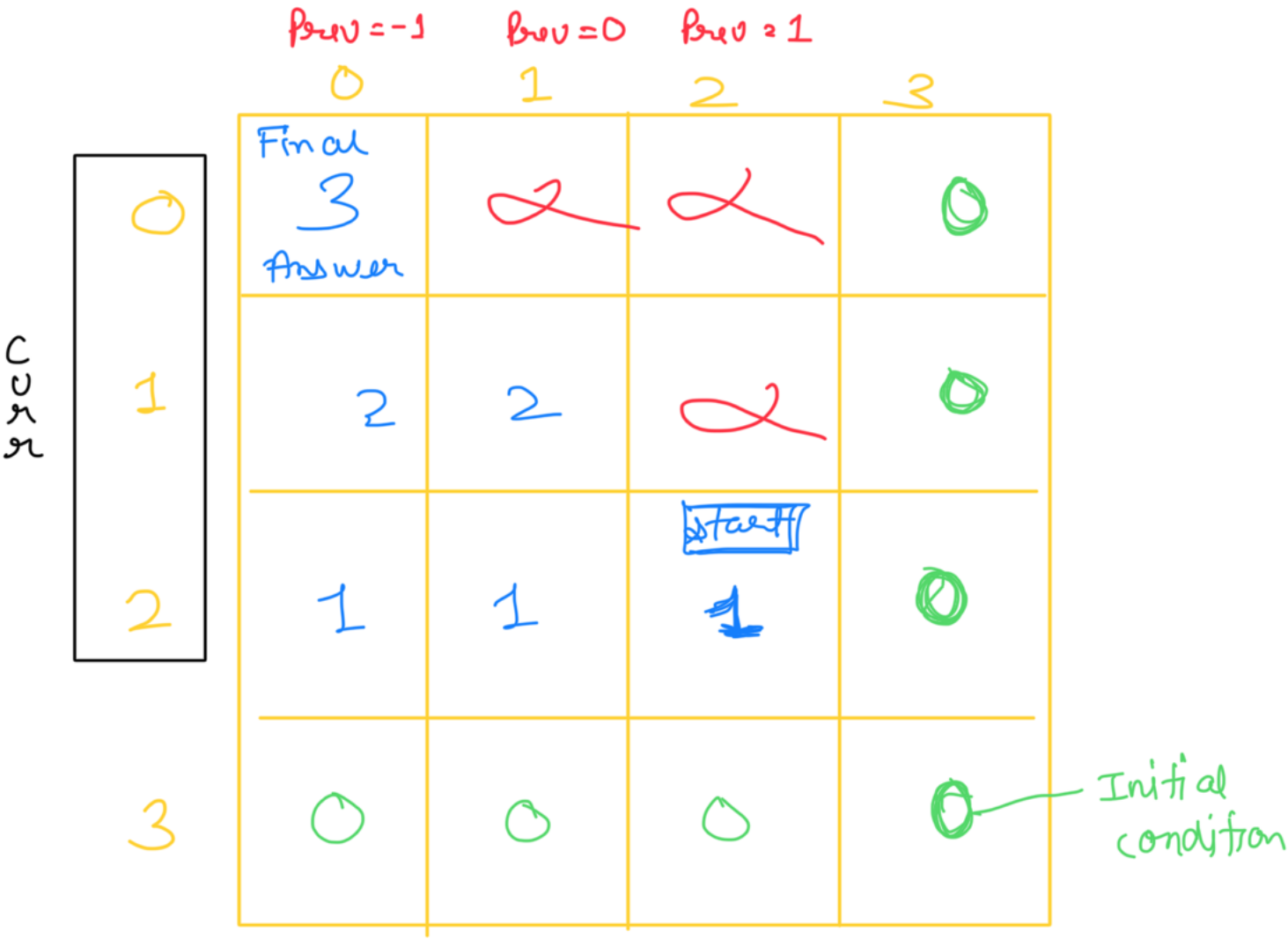
[1, 2, 3] (0, 1, 2) n = 3

curr = 0 to 2 : 0 1 2

prev = −1 to 1 : −1 0 1

↓ ↓ ↓

0 1 2



row → element starting This index can be included

column → Values above this can be included

exclude = db[curr+1][prev+1]

row

offsetting prev value by 1, so we need to add 1

include = 1 + dp[curr + 1][curr + 1]

CODE: i = curr j = prev

```
def lengthofLIS(nums):
    n = len(nums)
    dp = [[0]*(n+1) for _ in range(n+1)]
    # start filling from the end
    for i in range(n-1, -1, -1):
        for j in range(i, -1, -1):
            exclude = dp[i+1][j]
            include = 0
            if j-1 == -1 or nums[i] > nums[j-1]:
                include = 1 + dp[i+1][i+1]
            dp[i][j] = max(exclude, include)
    return dp[0][0]
```

* Calculation using 1-D array.

Given array :

0	1	2	3	4	5	6
1	2	1	4	3	4	5

dp[i] = length(LIS) where element at current index is included

→ Recurrence Relation

if nums[i] > nums[j] and dp[j]+1 > dp[i]

→ dp[i] = dp[j]+1

CODE :

```
def length of LIS (nums) :
    n = len (nums)
    dp = [1] * n
    max = 1
    for i in range (n):
        for j in range (i):
            if nums[i] > nums[j] and dp[j]+1
                                    > dp[i]
                dp[i] = dp[j+1]
            if dp[i] > max :
```

```
        max = dp[i]

    return max
```

* LIS with binary Search

[9, 2, 7, 3, 4, 10]

→ if element > than last included element then include

→ else find smallest included element that is ≥ new element and replace it with it

[2, 3, 4, 10]

* This does not always gives LIS → But we need only length

Ex : [2, 3, 4, 1]

[1, 3, 4] → Length 3 but not the actual subsequence

* We will be using modified version of binary search that finds last number ≥ num

CODE :

```
def lengthofLIS(nums):
```

```
def binarysearch(sub, num):
    left = 0
    right = len(sub)
    while left<right:
        mid = (left+right) // 2
        if sub[mid] < num
            left=mid+1
        else:
            right = mid
    return left

sub = [nums[0]]
n = len(nums)
for num in nums[1:]:
    if num > sub[-1]:
        sub.append[num]
    else:
        index = binary search (sub, num)
        sub[index] = num
return len(sub)
```

Question: Max length of pair chain

Input: [[1,2], [2,3], [3,4]]

Output: 2

Explanation : [1,2] → [3,4]

We will solve it using LIS tabulation approach.

CODE :

```
def findLongestChain(pairs):
    pairs.sort()
    n = len(pairs)

    dp = [1]*n
    max = 1

    for i in range(n):
        for j in range(i):
    # i = curr_index and j = prev_index
            if pairs[i][0] > pairs[j][1] and
                dp[j]+1 > dp[i]:
                dp[i] = dp[j]+1
        if dp[i] > max:
            max = dp[i]

    return Max
```

QUESTION : Russian Doll Envelopes

Ex : [[5,4],[6,4],[6,7],[2,3]]

Input : [[5,4],[6,7],[2,3]]

Output : 3

Explanation : [2,3], [5,4], [6,7]

Two approach : LIS Tabulation
LIS binary search.

Tabulation Approach

CODE :

```
def maxEnvelopes(envelopes):
    envelopes.sort(key = lambda x: x[x[0], x[-1]])
    n = len(envelopes)
    dp = [1]*n
    max = 1
    for i in range(1,n):
        for j in range(i):
            if envelopes[i][1] > envelopes[j][i]
                and dp[j]+1 > dp[i]
                dp[i] = dp[j]+1
        if dp[i] > max:
            max = dp[i]
    return max
```

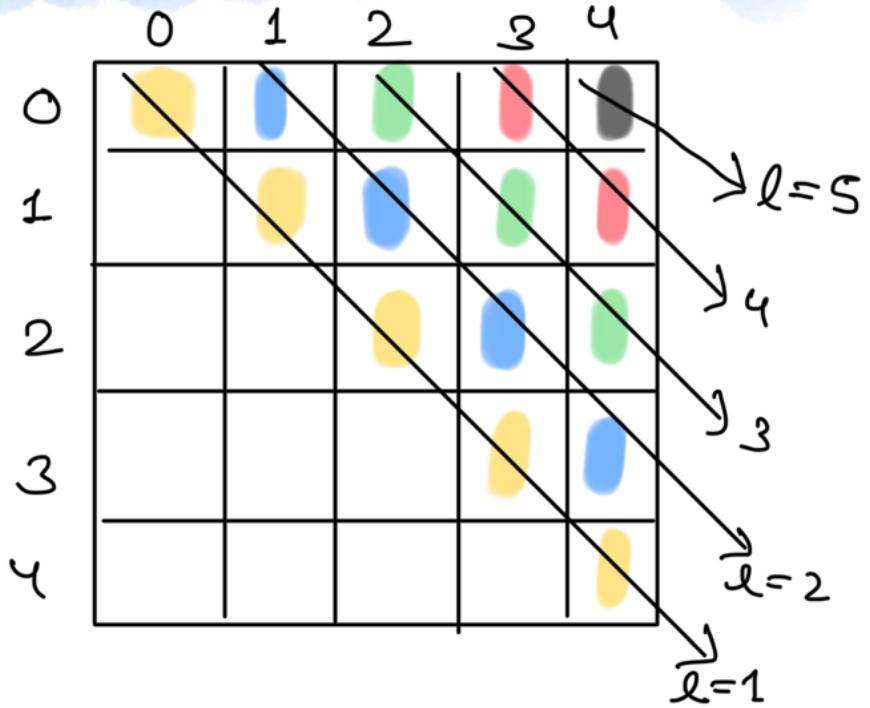
Binary Search Approach

```
def maxEnvelopes(envelopes):
    envelopes.sort(key = lambda x: x(x[0], x[-1]))
    n = len(envelopes)
    def binarysearch(sub, num):
        left = 0
        right = len(sub) - 1
        while left < right:
            mid = (left + right) // 2
            if sub[mid] < num:
                left = mid + 1
            else:
                right = mid
        return left
    sub = [envelopes[0][1]]
    for i in range(1,n):
        num = envelopes[i][1]
        if num > sub[-1]:
            sub.append(num)
```

```
else :
    index = binarysearch(sub, num)
    sub[index] = num

return len(sub)
```

GAP STRATEGY :



QUESTION : Palindromic Substrings

Input : aaba

Output : 6

Explanation : a, a, b, a , aa, aba

1) abba → check i+1

j - 1

2) aaba → check i+1 , j

3) abaa → check i , j - 1

4) "a" → is a palindrome

5) "aa" → is a palindrome

		0 p	1 q	2 p	3 k	4 p	5 s
0	p	T	F	T	F	F	F
1	q	∝	T	F	F	F	F
2	p	∝	∝	T	F	T	F
3	k	∝	∝	∝	T	F	F
4	p	∝	∝	∝	∝	T	F
5	s	∝	∝	∝	∝	∝	T

row represents start from : index

column represents ends at : index

isPalindrome (i+1, j)
isPalindrome (i, j-1)

0	1	2	3	4	5
p	q	p	r	p	s

CODE :

```
def countSubstrings(s):

    n = len(s)
    dp = [[-1] * n for _ in range(n)]

    def helper(i, j):

        if i == j:
            dp[i][j] = True
            return dp[i][j]

        if dp[i][j] != -1
            return dp[i][j]

        helper(i+1, j)
        helper(i, j-1)
```

```
        if s[i] == s[j] and (j == i+1 or
                             helper(i+1, j-1)):
            dp[i][j] = True
        else:
            dp[i][j] = False
        return dp[i][j]
    helper(0, n-1)
    for l in range(1, n+1)
        for i in range(n-l+1)
            j = i+l-1
            if dp[i][j] == True
                res += 1
    return res
```

TABULATION APPROACH

```
def countSubstrings(s):
    n = len(s)
    dp = [[-1] * n for _ in range(n)]
    res = 0

    for l in range(1, n+1)
        for i in range(n-l+1):
            j = i+l-1

            if i == j:
```

```
            dp[i][j] = True
            res += 1
        elif s[i] == s[j] and (j == i+1 or
                               dp[i+1][j-1]):
            dp[i][j] = True
            res += 1
        else:
            dp[i][j] = False
    return res
```

QUESTION : 5. Longest Palindromic Substring

Input : "babad"

Output : bab or aba
both are considered right

Same approach as previous question but with slight tweak.

CODE :

```
def longestPalindrome(s):
    n = len(s)
    dp = [[-1] * n for _ in range(n)]
    longest = ''

    for l in range(1, n+1):
        for i in range(n-l+1):
            j = i+l-1
```

```
            if i == j:
                dp[i][j] = True
            elif s[i] == s[j] and (j == i+1 or
                                   dp[i+1][j-1]):
                dp[i][j] = True
            else:
                dp[i][j] = False
            if dp[i][j]:
                longest = s[i:j+1]
        return longest
```

516. Longest Palindromic Subsequence

Input : s = "bbbab"

Output : 4

Explanation : "bbbb"

CODE :

```
def longestPalindromeSubseq(s):
    n = len(s)
    dp = [[0]*n for _ in range(n)]

    for l in range(1, n+1):
        for i in range(n-l+1):
```

```
        j = i + l - 1
        if i == j:
            dp[i][j] = 1
        elif a[i] == a[j]:
            dp[i][j] = 2 + dp[i+1][j-1]
        else:
            dp[i][j] = max(dp[i+1][j],
                           dp[i][j-1])
return dp[0][n-1]
```