Lesson:

Dynamic Programming-2

PW SKILLS

Pre Requisites:

- Basics of Dynamic Programming
- Top-down & Bottom-up Approach

List of concepts involved :

- Minimum path sum
- Longest common subsequence
- Longest increasing subsequence
- Matrix chain multiplication

Minimum Path Sum :

Q1. Given a triangle array, return the minimum path sum from top to bottom.For each step, you may move to an adjacent number of the row below. More formally, if you are on index i on the current row, you may move to either index i or index i + 1 on the next row.

Example 1:
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:
2
3 4
6 5 7
4 1 8 3
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).

Example 2:
Input: triangle = [[-10]]
Output: -10

Example 3:
Input : triangle = [[5] , [-3 , 4] , [6 , -8 , 10] , [3 , 100 , 200 , 1]]
Output : 20

Solution :

Code : https://pastebin.com/M6jp5REF

Output :

```
The desired output is : 11
```

Approach :

1. As we are now satisfied that greedy won't work here as we can pick a bigger element now to get much smaller elements in future, thus the only way to solve this problem is to try all possible combinations.
2. At each step we have two choices thus there will be total 2^level possible combination which will be very expensive to compute
3. Are all these calculations unique? like if you are at cell [i][j] then no matter which path you have taken to come there, the optimal path from this cell to the bottom row will be the same
4. as we know that the position[i][j] can uniquely define a sub-problem thus let's define our dp state with this

i.e.,
- dp[i][j] = what is the min path you can get if you start from this cell and want to reach the bottom level.

5. let's try to define the transition now,
   as we know that if you start from cell [i][j], you will have to either go to cell[i+1][j] or cell[i+1][j+1]
   thus, dp[i][j] = triangle[i][j] + min(dp[i+1][j], dp[i+1][j+1])

6. let's think about the base case now,
   as our defn of dp[i][j] = min path from this cell to last level
   thus dp[last_level][j] = triangle[last_level][j] for all j.

Longest common subsequence

Q2. Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0.
A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".
A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

Solution :
Code : https://pastebin.com/Cun0Fcx9

Output :

```
The desired output is : 3
```

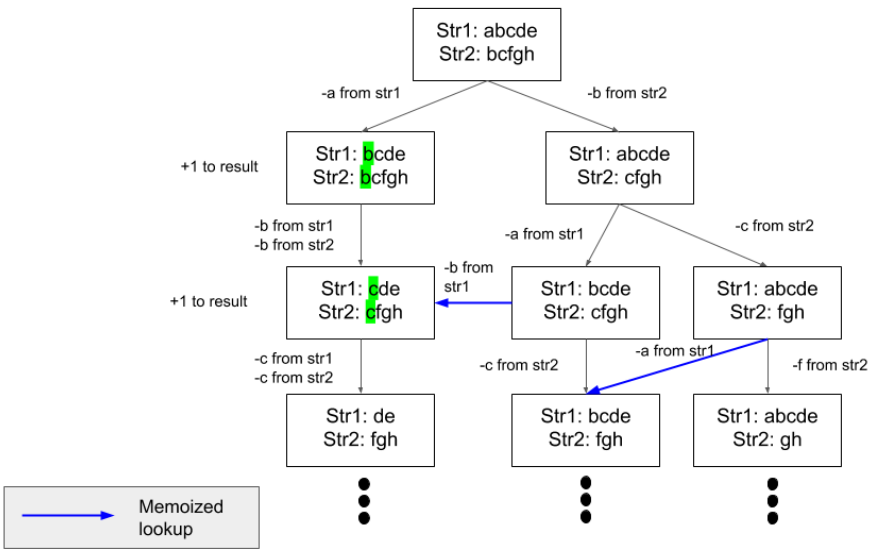
Approach :
- First string: str1, Second string: str2.

Assume you are at position ptr1 for str1 and position ptr2 for str2.

- **Step 1:** Compare str1[ptr1] and str[ptr2]. There can be two cases arising from this comparison:
  - **Case 1:** They are the same
    - If they are the same, it is simple, you know that there is 1 good subsequence that you can add up to your answer. Then you increment ptr1 and ptr2. Go back to Step 1.
  - **Case 2:** They are not the same.

- If that's the case, now you have to explore two options:
  - **Op 1:** You increment ptr1, and go back to Step 1.
  - **Op 2:** You increment ptr2, and go back to Step 1.
- You do this until either of your pointers reach to the end of either of the strings.
- For either of the options in Case 2, there will be a lot of overlapping cases.
- In the diagram,'-a from str1' or '-b from str 2'. refers to "removing a from str1" and "removing b from str2", respectively. It's similar to incrementing the pointers of respective strings.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	**0**	0	0	0	0	0	0	0
1	M	0	0	0	0	0	0	1	1
2	Z	0	**1**	1	1	1	1	1	1
3	J	0	1	1	**2**	2	2	2	2
4	A	0	1	1	2	2	2	2	2
5	W	0	1	1	2	**3**	3	3	3
6	X	0	1	1	2	3	3	3	**4**
7	U	0	1	2	2	3	3	3	4



Longest Increasing Subsequence

Q3. Given an integer array nums, return the length of the longest strictly increasing subsequence.

Example 1:
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:

Input: nums = [0,1,0,3,2,3]
Output: 4

Example 3:
Input: nums = [7,7,7,7,7,7,7]
Output: 1

Solution :
Code : https://pastebin.com/WFe4tirx

Output :

```
The desired output is : 5
```

Approach :

- In dp array we're storing the LIS up to that number, initially all 1 (default case)
- Then we start iterating from the last index and run another loop inside this, this inner loop will help us to find the LIS on the right of the current element.
- So when iterating this inner loop we'll check that if the element at jth index is larger than the element at the current index, if it is then we'll store the max of (dp[i], 1+dp[j]).
- At last for each iteration of the outer loop we'll check and update the max LIS.
- Return max value.

Matrix chain multiplication

Q4. Given the dimension of a sequence of matrices in an array arr[], where the dimension of the ith matrix is (arr[i-1] * arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

Examples:
Input: arr[] = {40, 20, 30, 10, 30}
Output: 26000
Explanation: There are 4 matrices of dimensions 40×20, 20×30, 30×10, 10×30.
Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by
putting parentheses in the following way (A(BC))D.
The minimum is 20*30*10 + 40*20*10 + 40*10*30

Input: arr[] = {1, 2, 3, 4, 3}
Output: 30
Explanation: There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3.
Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by
putting parentheses in the following way ((AB)C)D.
The minimum number is 1*2*3 + 1*3*4 + 1*4*3 = 30

Input: arr[] = {10, 20, 30}
Output: 6000
Explanation: There are only two matrices of dimensions 10×20 and 20×30.
So there is only one way to multiply the matrices, cost of which is 10*20*30

Solution :

Code : https://pastebin.com/uz5KVMrq

Output :

```
The desired output is : 36
```

Approach :

- Two matrices of size m*n and n*p when multiplied, they generate a matrix of size m*p and the number of multiplications performed are m*n*p.
- Now, for a given chain of N matrices, the first partition can be done in N-1 ways. For example, sequences of matrices A, B, C and D can be grouped as (A)(BCD), (AB)(CD) or (ABC)(D) in these 3 ways.
- So a range [i, j] can be broken into two groups like {[i, i+1], [i+1, j]}, {[i, i+2], [i+2, j]}, . . . , {[i, j-1], [j-1, j]}.
- Each of the groups can be further partitioned into smaller groups and we can find the total required multiplications by solving for each of the groups.
- The minimum number of multiplications among all the first partitions is the required answer.
- Optimal Substructure: In the above case, we are breaking the bigger groups into smaller subgroups and solving them to finally find the minimum number of multiplications. Therefore, it can be said that the problem has optimal substructure properties.
- Overlapping Subproblems: We can see in the recursion tree that the same subproblems are called again and again and this problem has the Overlapping Subproblems property.
- So the Matrix Chain Multiplication problem has both properties of a dynamic programming problem. So recomputing the same subproblems can be avoided by constructing a temporary array dp[][] in a bottom up manner.
- Follow the below steps to solve the problem:
- Build a matrix dp[][] of size N*N for memoization purposes.
- Use the same recursive call as done in the above approach:
- When we find a range (i, j) for which the value is already calculated, return the minimum value for that range (i.e., dp[i][j]).
- Otherwise, perform the recursive calls as mentioned earlier.
- The value stored at dp[0][N-1] is the required answer.

Dynamic Programming Solution for Matrix Chain Multiplication using Tabulation (Iterative Approach):

- In iterative approach, we initially need to find the number of multiplications required to multiply two adjacent matrices. We can use these values to find the minimum multiplication required for matrices in a range of length 3 and further use those values for ranges with higher lengths.
- Build on the answer in this manner till the range becomes [0, N-1].
- Follow the steps mentioned below to implement the idea:
- Iterate from l = 2 to N-1 which denotes the length of the range:
- Iterate from i = 0 to N-1:
- Find the right end of the range (j) having l matrices.
- Iterate from k = i+1 to j which denotes the point of partition.
- Multiply the matrices in range (i, k) and (k, j).
- This will create two matrices with dimensions arr[i-1]*arr[k] and arr[k]*arr[j].
- The number of multiplications to be performed to multiply these two matrices (say X) are arr[i-1]*arr[k]*arr[j].
- The total number of multiplications is dp[i][k]+ dp[k+1][j] + X.
- The value stored at dp[1][N-1] is the required answer.

Code : https://pastebin.com/XekH1iXZ

Upcoming Class Teasers:

- Interview Problems on DP