Lesson:

Dynammic Programming-2

Pre Requisites:

- Recursion
- Basic JAVA syntax

List of concepts involved :

- Longest common subsequence
- Longest increasing subsequence
- Matrix chain multiplication
- Catalan Number — Unique BST
- Subset sum equal to K

Longest common subsequence

Q1. 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 : LP_code1.java

Output :

```
The desired output is : 4
```

Approach :

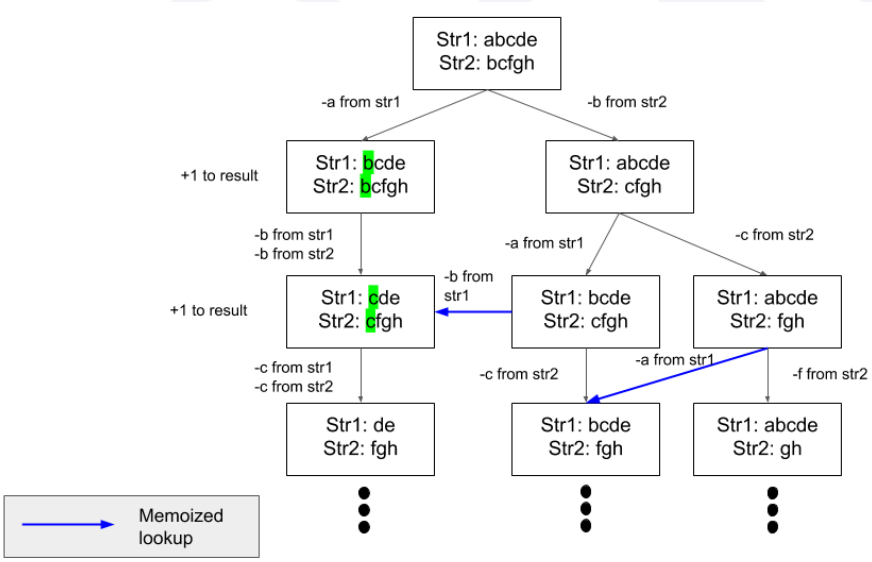
- First string: str1, Second string: str2.

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

Q2. 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 : LP_Code2.java

Output :

```
The desired output is : 4
```

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

Q3. 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 : LP_Code3.java

Output :

```
Minimum number of multiplications is 18
```

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 : LP_Code4.java

Catalan Number — Unique BST

Q4. Catalan numbers are defined as a mathematical sequence that consists of positive integers, which can

be used to find the number of possibilities of various combinations.

The nth term in the sequence denoted Cn, is found in the following formula: (2n)! / (n + 1)!* n!

The first few Catalan numbers for n = 0, 1, 2, 3, … are : 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …
Given a number n. Print the nth catalan number.

Examples:
Input: n = 6
Output: 132

Input: n = 8
Output: 1430

Solution :
Code : LP_Code5.java

Output :

```
The desired output is : 208012
```

Approach :
- Create an array catalan[] for storing ith Catalan number.
- Initialize, catalan[0] and catalan[1] = 1
- Loop through i = 2 to the given Catalan number n.
- Loop through j = 0 to j < i and Keep adding value of catalan[j] * catalan[i – j – 1] into catalan[i].
- Finally, return catalan[n].

Subset sum equal K

Q5. Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Example:
Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that adds up to 30.

Solution :
Code : LP_Code6.java

Output :

```
Found a subset with given sum
```

Approach :

- we will create a 2D array of size (arr.size() + 1) * (target + 1) of type boolean. The state DP[i][j] will be true if there exists a subset of elements from A[0....i] with sum value = 'j'. The approach for the problem is:
- if (A[i-1] > j)
  - DP[i][j] = DP[i-1][j]
- else
  - DP[i][j] = DP[i-1][j] OR DP[i-1][j-A[i-1]]
- This means that if current element has value greater than 'current sum value' we will copy the answer for previous cases
- And if the current sum value is greater than the 'ith' element we will see if any of previous states have already experienced the sum='j' OR any previous states experienced a value 'j – A[i]' which will solve our purpose.
- The below simulation will clarify the above approach:

set[]={3, 4, 5, 2}
target=6

	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
3	T	F	F	T	F	F	F
4	T	F	F	T	T	F	F
5	T	F	F	T	T	T	F
2	T	F	T	T	T	T	T

For example :
set[]={3, 4, 5, 2}
sum=9
(x, y)= 'x' is the left number of elements,
'y' is the required sum