a) Longest Common Subsequence

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. In LCS, we have to find the Longest Common Subsequence that is in the same relative order.

In the give question, let the name of Group Member 1 is **D**anish and Group member 2 is S**a**lman, so:

X: {B, D, C, A, B, A} and Y: {A, B, C, B, D, A, B}

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

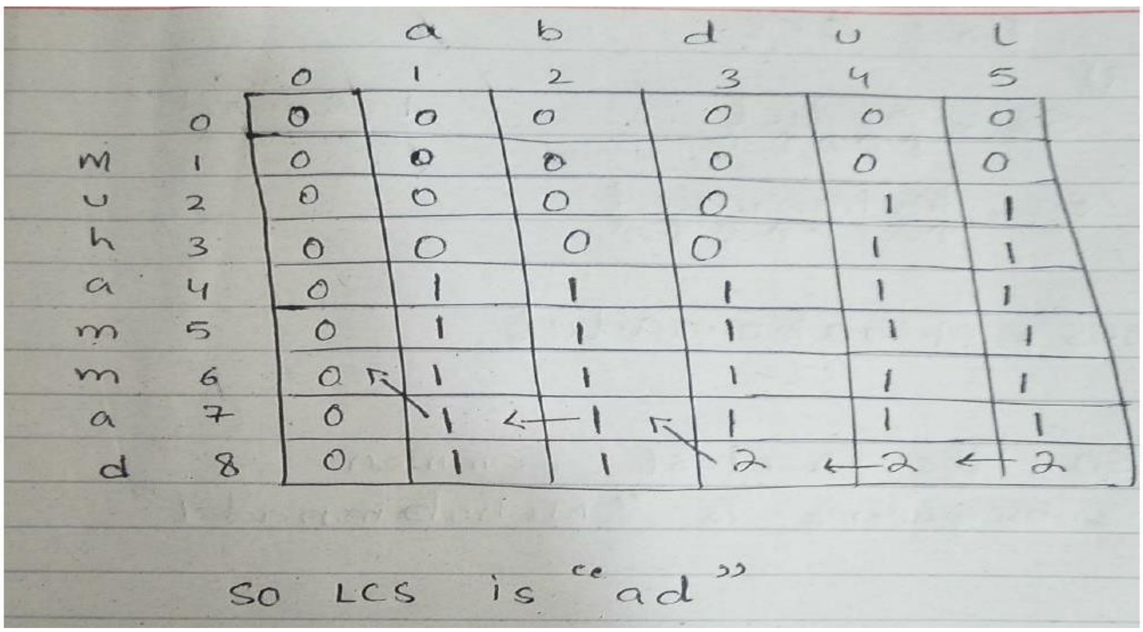
(b) Shortest-Common-Supersequence

Example Problem

S1 = “abdul”

S2 = “muhammad”

Now we first need to find the LCS of both strings by dynamic programming approach.



			a	b	d	u	l
		0	1	2	3	4	5
	0	0	0	0	0	0	0
m	1	0	0	0	0	0	0
u	2	0	0	0	0	1	1
h	3	0	0	0	0	1	1
a	4	0	1	1	1	1	1
m	5	0	1	1	1	1	1
m	6	0	1	1	1	1	1
a	7	0	1	1	1	1	1
d	8	0	1	1	2	2	2

So LCS is “ad”

Now for the shortest common supersequence we need to find the following;

1) Length of shortest common supersequnce

   Length of SCS =(length of S1 + length of S2) - Length of LCS

   =(5+8) - 2
   = 11

   So in our example the length of SCS will be **11**

Find SCS :

Note:

1. Add LCS characters only once .
2. Assume first common character belongs to LCS character.
3. Add non LCS characters in order of either S1 then S2 or vice versa.

S1= a b d u l

S2 = m u h a m m a d

LCS = a d

SCS = muhabmmadul

So “muhabmmadul” is the SCS in our example of length **11**

Algorithm

```
int p1=0,p2=0;
       for(char c: lcs)
           while(str1[p1]!=c)   //Add all non-LCS chars from str1
               ans += str1[p1++];
           while(str2[p2]!=c)   //Add all non-LCS chars from str2
               ans += str2[p2++];
           ans += c;    //Add LCS-char and increment both ptrs
           ++p1
           ++p2
Return ans
```

c) Longest Increasing subsequence

Algo Assignment #3

Date:

Roll No of Member#1 = 18K-0264

Roll No of Member#2 = 18K-0350

Problem# longest increasing Subsequence.

Q) 4 10 2 0 20

Algorithm used:-

$L(i) = 1 + \max(L(j))$ where $0<j<1$ and array[j] < array[i];

$L(i) = 1$, if no j exists

Solution:- making new array of same size with all values 1

iteration #1(i) i=1, j=0 => array[j] < array[i], add 1, j++, i==j, new iteration

1	1	1	1	1

1	2	1	1	1

iteration #2(i) i=2, j=0 => array[j] > array[i], j++

iteration #2(ii) i=2, j=1 => array[j] > array[i], j++, i==j, new iteration

iteration #3(i) i=3, j=0 => array[j] > array[i], j++

iteration #3(ii) i=3, j=1 => array[j] > array[i], j++

iteration #3(iii) i=3, j=2 => array[j] > array[i], j++, i==j, new iteration

iteration #4(i) i=4, j=0 => array[j] < array[i], add 1 to j, j++

1	2	1	1	2

iteration #4(ii) i=4, j=1 => array[j] < array[i], add 1 to ith position pick from jth position, j++

1	2	1	1	3

iteration #4(iii) i=4, j=2 => array[j] < array[i] but we didn't add because already max value present, j++

Page No.

Date:

iteration #4(iv) i=4, j=3=> array[j] < array[i], we didn't add because max value already present, j++, i==j new iteration

So longest common subsequence is = $\boxed{3}$

d) Levenshtein-Distance

For Levenshtein-distance (edit-distance) problem, str1 = “KITTEN”, str2 = “SITTING”

For Levenshtein-distance (edit-distance) problem, str1 = “KITTEN”, str2 = “SITTING”

To calculate the Levenshtein distance (edit distance) between the two strings "KITTEN" and "SITTING", we can use a dynamic programming approach. This distance represents the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other.

Step 1: Initialize the DP Table

1. **Let m be the length of (KITTEN), which is 6.**
2. **Let n be the length of (SITTING), which is 7.**

Create a (m + 1) × (n + 1) table, initialized to zero.

Step 2: Base Cases

Fill in the first row and the first column:

dp[i][0] = i (deleting all characters of str1)

dp[0][j] = j (inserting all characters of str2)

Step 3: Fill the DP Table

Now we fill in the table based on the following rules: If the characters are the same:

dp[i][j] = dp[i − 1][j − 1]

If the characters are different:

dp[i][j] = 1 + min(dp[i − 1][j], dp[i][j − 1], dp[i − 1][j − 1]) dp[i − 1][j] for deletion

dp[i][j − 1] for insertion

dp[i − 1][j − 1] for substitution

Step 4: Fill in the Values

		S	I	T	T	I	N	G
	0	1	2	3	4	5	6	7
K	1	1	2	3	4	5	6	7
I	2	2	1	2	3	4	5	6
T	3	3	2	1	2	3	4	5
T	4	4	3	2	1	2	3	4
E	5	5	4	3	2	2	3	4
N	6	6	5	4	3	3	2	3

The value in the bottom-right corner of the table dp[6][7] gives us the Levenshtein distance, which is 3.

e) Matrix Chain Multiplication

```
Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}
Output: 6000
There are only two matrices of dimensions 10x20 and 20x30. So there
is only one way to multiply the matrices, cost of which is 10*20*30
```

```
#include <bits/stdc++.h>
using namespace std;


// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;


    // place parenthesis at different places
    // between first and last matrix, recursively
    // calculate count of multiplications for
    // each parenthesis placement and return the
```

```
        // minimum count
        for (k = i; k < j; k++)
        {
            count = MatrixChainOrder(p, i, k)
                    + MatrixChainOrder(p, k + 1, j)
                    + p[i - 1] * p[k] * p[j];

            if (count < min)
                min = count;
        }

        // Return minimum count
        return min;
    }

    // Driver Code
    int main()
    {
        int arr[] = { 1, 2, 3, 4, 3 };
        int n = sizeof(arr) / sizeof(arr[0]);

        cout << "Minimum number of multiplications is "
             << MatrixChainOrder(arr, 1, n - 1);
    }
```

f) Knapsack Problem

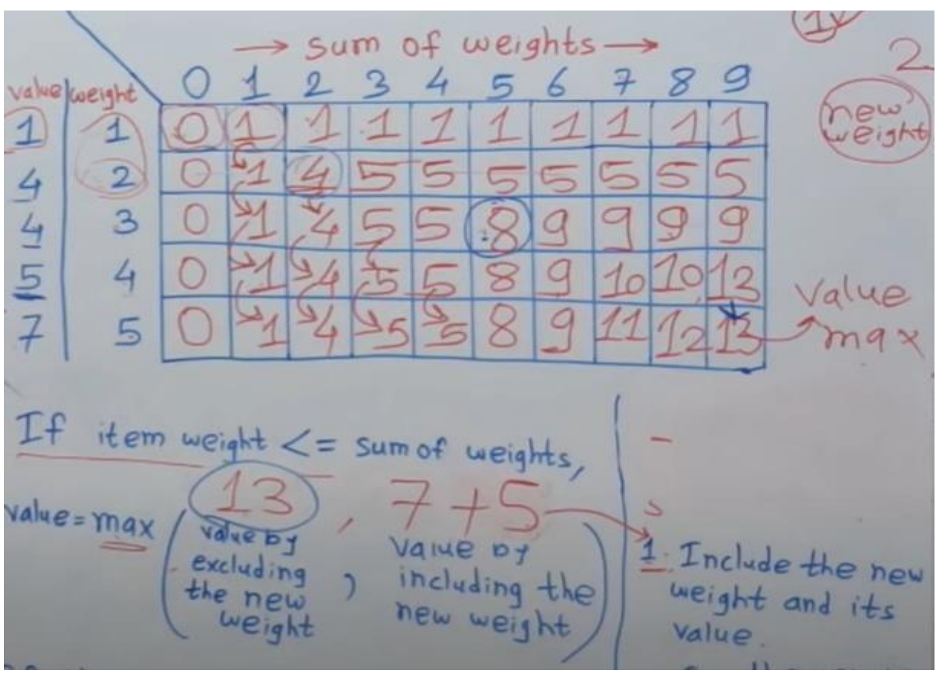
Given a set ‘S’ of ‘n’ items,

such that each item i has a positive value vi and positive weight w_i

The goal is to find maximum-benefit subset that does not exceed the given weight W.

Algorithm

```
Knapsack(j,w)
  for i ← 0 to n
          M[i,0] ← 0
 for w ← 0 to W
          M[0,w] ← 0
 for j ← 1 to n
          for w ← 0 to W
                  if wj > w
                       M[j,w] = M[j – 1, w))
                  else M[j,w] ← MAX(vj + M[j – 1, w – wj],
M[j – 1, w])
return M[n,W]
```



The maximum benefit/value = 13, The selected items’ indexes are: s_i = [0,1,1,1,0]

g) Partition Problem

Given a set of positive integers, check if it can be divided into two subsets with equal sum.

- First. The sum of all elements in the set is calculated. If sum is odd, we can't divide the array into two sets. If sum is even, check if a subset with sum/2 exists or not.

Let the first three alphabets converted to numbers:

Group member 1: Abid 1,2,9

Group member 2: Adam 1,4,1

Set= 1,2,9,1,4,1

Sum=18 (Since the sum is even, so partition might be possible)
Sum/2=9 We will make table of size 9, and if the last element results in True, then the partition is possible, otherwise not.

This somewhat resembles to the 0/1 knapsack problem.

	0	1	2	3	4	5	6	7	8	9
0	**T**	F	F	F	F	F	F	F	F	F
1	**T**	**T**	F	F	F	F	F	F	F	F
2	**T**	**T**	**T**	F	F	F	F	F	F	F
9	**T**	**T**	**T**	F	F	F	F	F	F	**T**
1	**T**	**T**	**T**	**T**	F	F	F	F	F	**T**
4	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
1	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	$\boxed{\textbf{T}}$

The last box of the table is true, that shows that the partition is possible.

The result showed that set can be partitioned into S1 = {9}, and S2 = {1,2,1,4,1}, with each subset having a sum of 9. However, note that the size of sets are not equal.

h) Rod Cutting Problem

Rod cutting problem is a type of allocation problem. Allocation problem involves the distribution of resources among the competing alternatives in order to minimize the total costs or maximize total return (profit).

In the rod cutting problem we have given a rod of length n, and an array that contains the prices of all the pieces smaller than n, determine the maximum profit you could obtain from cutting up the rod and selling its pieces.

Length[] = {1,2,3,4,5,6,7,8}

price[] = {1,5,8,9,10,16,18,20}

Rod Length: 8

Rod Length (i)	Cut 0	Cut 1	Cut 2	Cut 3	Cut 4	Cut 5	Cut 6	Cut 7	Cut 8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	1	5	6	10	11	15	16	20
3	0	1	5	8	10	13	15	18	20

4	0	1	5	8	10	13	15	18	20
5	0	1	5	8	10	13	15	18	20
6	0	1	5	8	10	13	16	18	21
7	0	1	5	8	10	13	16	18	21
8	0	1	5	8	10	13	16	18	22

Mximum Profit = 22

Selected Pieces = 8 pieces of Rod Length 2, to get maximum profit for rod length 22.

i) Coin Change Making Problem

- The coin change problem addresses the question of finding the minimum number of coins (of certain denominations) that add up to a given amount of money. It is special case of integer knapsack problem.
- Let C[p] be the minimum number of coins of denominations d1, d2, . . . , dk needed to make change for p cents.
- $$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_i \{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

```
CHANGE(d, k, n)
C[0] ← 0
for p ← 1 to n
        min ← ∞
        for i ← 1 to k
            if d[i] ≤ p then
                if 1 + C[p − d[i]] < min then
                    min ← 1 + C[p − d[i]]
                    coin ← i
        C[p] ← min
        S[p] ← coin
return C and S
```

```
MAKE-CHANGE(S, d, n)
while n > 0
    Print S[n]
    n ← n − d[S[n]]
```

	Coin	Weight													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
All coins 1,5,6, and 8	8	0	1	2	3	4	1	1	2	1	2	2	2	2	2
Coin 1,5 and 6	6	0	1	2	3	4	1	1	2	3	4	2	2	3	4
Coin 1 and 5	5	0	1	2	3	4	1	2	3	4	5	2	3	4	5
Coin1	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
No coin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

j) Word Break

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.

We will try to search from the left of the string to find a valid word when a valid word is found, we will search for words in the next part of that string.

For Word Break Problem, S = {i, like, ice, cream, icecream, mobile, apple}

INPUT:

Ilikeapple

OUTPUT:

i like apple

		i	l	i	k	e	a	p	p	l	e
		0	1	2	3	4	5	6	7	8	9
i	0	**T**	F	F	F	**T**	F	F	F	F	**T**
l	1		F	F	F	**T**	F	F	F	F	F
i	2			**T**	F	F	F	F	F	F	F
k	3				F	F	F	F	F	F	F
e	4					F	F	F	F	F	F
a	5						F	F	F	F	**T**
p	6							F	F	F	F
p	7								F	F	F
l	8									F	F
e	9										F

Q2: Take any real-world daily life task and write an algorithm to solve that using dynamic programming approach that is mapped on one of the problems above. For example, You want to plan meals for a week while maximizing nutritional value within a budget. Each meal has a cost and a nutritional value. The goal is to select meals that fit within your budget and maximize total nutritional value. Also dry run the algorithm.

Solution:

To solve the problem of planning meals for a week while maximizing nutritional value within a budget, we can employ a method similar to the **0/1 Knapsack problem**. Below is a structured approach to tackle this issue.

Step 1: Define the Problem

- **Goal**: Select meals that fit within a budget while maximizing total nutritional value.
- **Inputs**:

- **Budget (B)**: Total amount available for meals.
- **Meals**: A list of meals, each with a cost and a nutritional value.

Step 2: Data Representation

Let's represent our meals as tuples of (cost, nutritional value). For example:

Meals:

1. Meal A: (cost: 5, nutritional value: 10)

2. Meal B: (cost: 3, nutritional value: 5)

3. Meal C: (cost: 4, nutritional value: 7)

4. Meal D: (cost: 6, nutritional value: 12)

5. Meal E: (cost: 2, nutritional value: 3)

Step 3: Budget Constraint

Assume the budget for meals is:

Budget (B): 15

Step 4: Algorithm Design

We can use a dynamic programming approach:

1. Create a table dp where dp[i][j] represents the maximum nutritional value achievable with the first i meals and a budget j.
2. Initialize the table with zeros.
3. Iterate through each meal and update the table based on whether to include the meal or not.

Step 5: Dynamic Programming Solution

Implementation

```
def customalgo(meals, budget):
  n = len(meals)
  dp = [[0] * (budget + 1) for _ in range(n + 1)]
  for i in range(1, n + 1):
    cost, value = meals[i - 1]
    for j in range(budget + 1):
      if cost <= j:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost] + value)
      else:
        dp[i][j] = dp[i - 1][j]
```

```
    return dp[n][budget]

max_nutritional_value = customalgo(meals, budget)

print("Maximum Nutritional Value:", max_nutritional_value)
```

Dry run will be similar to Knapsack.