Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 4

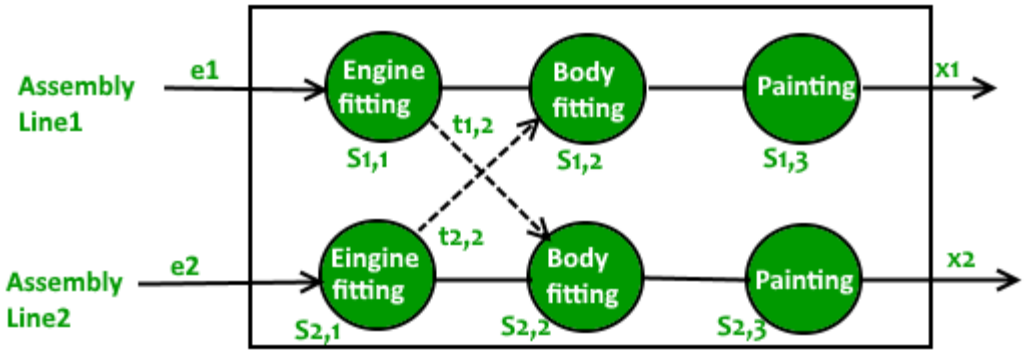
Aim – To implement Dynamic programming algorithms to solve Assembly line scheduling and Longest common subsequence(LCS) problems.

Details – Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping sub-problems and optimal substructure property. If any problem can be divided into sub-problems, which in turn are divided into smaller sub-problems, and if there are overlapping among these subproblems, then the solutions to these sub-problems can be saved for future reference.

Part 1 : Assembly line scheduling problem

Problem statement:

A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting, and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.



Pseudocode :

Algorithm Assembly-Line-Scheduling(a, t, e, x, n)

Input: Arrays a[1..2, 1..n] and t[1..2, 1..n-1] of nonnegative integers, arrays e[1..2] and x[1..2] of entry and exit times, and an integer n.

Output: The minimum time to get through the assembly line.

```
1. Let f[1..2, 1..n] and l[1..2, 1..n] be new arrays
2. f[1, 1] = e[1] + a[1, 1] and f[2, 1] = e[2] + a[2, 1]
3. for j = 2 to n
4.     for i = 1 to 2
5.         if f[1, j-1] + a[i, j] <= f[2, j-1] + t[2, j-1] + a[i, j]
6.             then f[i, j] = f[1, j-1] + a[i, j]
7.                 l[i, j] = 1
8.         else f[i, j] = f[2, j-1] + t[2, j-1] + a[i, j]
9.             l[i, j] = 2
10.    end for
11. end for
12. if f[1, n] + x[1] <= f[2, n] + x[2]
13.    then f* = f[1, n] + x[1]
14.        l* = 1
15. else f* = f[2, n] + x[2]
16.     l* = 2
```

17. return f* and l*

Here, f* and l* represent the fastest time to get through the factory and the line number used to exit from the nth station respectively.

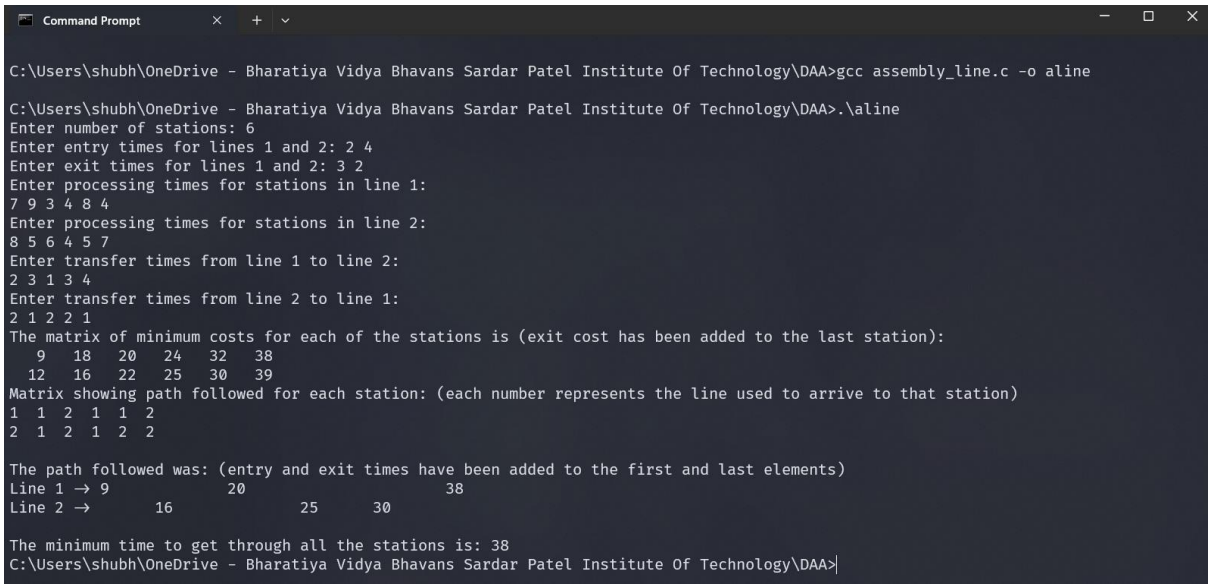
Source code(C language):

```c
#include<stdio.h>
#include<stdlib.h>

int main(){
    int n;
    printf("Enter number of stations: ");
    scanf("%d", &n);
    int a[n],b[n],ta[n-1],tb[n-1],e1,e2,x1,x2,path[2][n];
    printf("Enter entry times for lines 1 and 2: ");
    scanf("%d%d", &e1, &e2);
    printf("Enter exit times for lines 1 and 2: ");
    scanf("%d%d", &x1, &x2);
    printf("Enter processing times for stations in line
1:\n");
    for(int i=0;i<n;i++){
        scanf("%d", &a[i]);
    }
    printf("Enter processing times for stations in line
2:\n");
    for(int i=0;i<n;i++){
        scanf("%d", &b[i]);
    }
    printf("Enter transfer times from line 1 to line 2:\n");
    for(int i=0;i<n-1;i++){
        scanf("%d", &ta[i]);
    }
    printf("Enter transfer times from line 2 to line 1:\n");
    for(int i=0;i<n-1;i++){
        scanf("%d", &tb[i]);
    }
    a[0]+=e1;
    b[0]+=e2;
    path[0][0]=1;
    path[1][0]=2;
```

```
    for(int i=1;i<n;i++){
        if(a[i]+a[i-1]<a[i]+b[i-1]+tb[i-1]){
            a[i]+=a[i-1];
            path[0][i]=1;
        }
        else{
            a[i]+=b[i-1]+tb[i-1];
            path[0][i]=2;
        }
        if(b[i]+b[i-1]<b[i]+a[i-1]+ta[i-1]){
            b[i]+=b[i-1];
            path[1][i]=2;
        }
        else{
            b[i]+=a[i-1]+ta[i-1];
            path[1][i]=1;
        }
    }
    a[n-1]+=x1;
    b[n-1]+=x2;
    int follow=a[n-1]<b[n-1]?1:2;
    int f[n];
    f[n-1]=follow;
    for(int i=n-2;i>=0;i--){
        f[i]=path[f[i+1]-1][i+1];
    }

    printf("The matrix of minimum costs for each of the
stations is (exit cost has been added to the last
station):\n");
    for (int i = 0; i < n; i++){
        printf("%4d ",a[i]);
    }printf("\n");
    for (int i = 0; i < n; i++){
        printf("%4d ",b[i]);
    }printf("\n");
    printf("Matrix showing path followed for each station:
(each number represents the line used to arrive to that
station)\n");
    for(int i=0;i<2;i++){
        for(int j=0;j<n;j++){
            printf("%d  ",path[i][j]);
```

```
        }
        printf("\n");
    }
    printf("\n");

    printf("The path followed was: (entry and exit times have
been added to the first and last elements)\n");
    printf("Line 1 -> ");
    for(int i=0;i<n;i++){
        if(f[i]==1){
            printf("%d\t",a[i]);
        }
        else{
            printf("\t");
        }
    }
    printf("\n");
    printf("Line 2 -> ");
    for(int i=0;i<n;i++){
        if(f[i]==2){
            printf("%d\t",b[i]);
        }
        else{
            printf("\t");
        }
    }
    printf("\n");
    printf("\nThe minimum time to get through all the
stations is: %d",follow==1?a[n-1]:b[n-1]);

return 0;
}
```

Output:



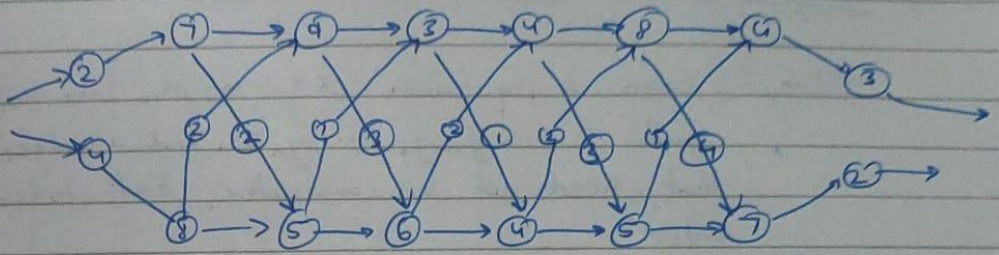
Conclusion:

- This dynamic programming solution executes in O(n) time complexity. The naïve recursive approach to solve this problem by tracing all of the paths would have O(2^n) time complexity.
- This solution satisfies the optimal substructure property as in the process of arriving to the final answer, we also found the optimal solutions to all of the all of the subproblems (the minimum time to arrive at any station).

Rough Working :

$f^* = \min (f[n] + x_1 , f_2[n] + x_2)$

objective function



$$F_1(j) = \begin{cases} e_1 + a_{1j} & \text{if } j = 1 \\ \min (F_1[j-1] + a_{1j} , f_2[j-1] + t_{2,j-1} + a_{1j}) \end{cases}$$

$$F_2(j) = \begin{cases} e_2 + a_{2j} & \text{if } j = 1 \\ \min (F_2[j-1] + a_{2j} , f_1[j-1] + t_{1,j-1} + a_{2j}) \end{cases}$$

Compute the optimal sol^n

First find optimal sol^n to subproblem and then use to find optimal sol^n to problem. for $j \geq 2$, each value $f_i[j]$ depends on $f_1[j-1]$ and $f_2[j-1]$

	1	2	3	4	5	6
$F_1[j]$	$F_1(1)$	$F_1(2)$	$F_1(3)$	$F_1(4)$	$F_1(5)$	$F_1(6)$
$F_2[j]$	$F_2(1)$	$F_2(2)$	$F_2(3)$	$F_2(4)$	$F_2(5)$	$F_2(6)$

9	18	20	24	32	35	→ 38 → exit
12	16	22	25	30	37	→ 39

Ans (38)

1	1	2	1	1	2	→
2	1	1	1	2	2	

Part 2 : Longest common subsequence problem

Problem Statement:

The longest common subsequence problem is the problem of finding the longest subsequence common to all sequences in a set of sequences. Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

Input –

1) Sizes of two sequences

2) Two sequences of maximum 50 characters e.g. X=“ABCDGH” and Y=“AEDFHR”

Output –

1) m[1..|X|][1..|Y|]=Two dimension matrix of Optimal Solutions (No. of longest sub-sequences) of two sequences

2) One of the longest sub-sequences (There may be more than one subsequence with the longest length). Ex. “ADH” in this case.

Pseudocode:

Algorithm Longest-Common-Subsequence(a, b, n, m)

Input: Sequences a[1..n] and b[1..m]

Output: The length of the longest common subsequence and one of the longest common subsequences.

1. Let arr[0..m, 0..n] be a new array
2. for i = 0 to n
3. arr[0][i] = 0
4. for i = 0 to m
5. arr[i][0] = 0
6. for i = 1 to m
7. for j = 1 to n

8. if a[j-1] == b[i-1]
9. then arr[i][j] = arr[i-1][j-1] + 1
10. else arr[i][j] = max(arr[i-1][j], arr[i][j-1])
11. end for
12. end for
13. Print the length of the longest common subsequence arr[m][n]
14. Let s[1..arr[m][n]] be a new sequence
15. i = m, j = n
16. while arr[i][j] > 0
17. if arr[i-1][j] == arr[i][j]
18. then i = i - 1
19. else s[arr[i][j]] = b[i-1]
20. i = i - 1
21. j = j - 1
22. end while
23. if arr[m][n] != 0
24. then Print the longest common subsequence s
25. else Print "No common subsequence"
26. return

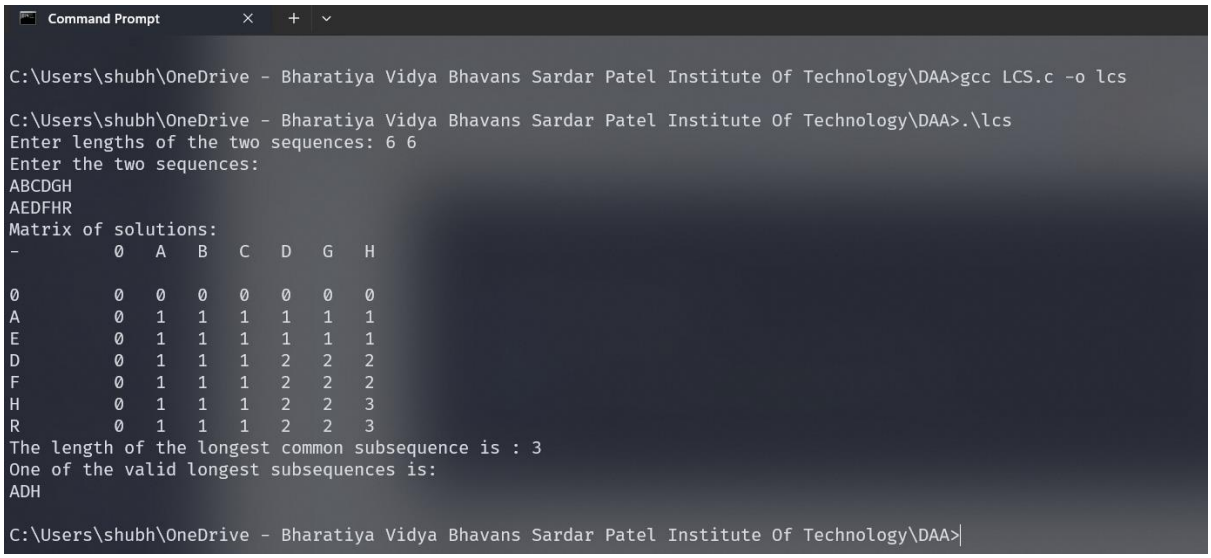
Source code (C language) :

```c
#include<stdio.h>
#include<stdlib.h>

int main(){
    int n,m;
    printf("Enter lengths of the two sequences: ");
    scanf("%d%d",&n,&m);
    char a[n+1],b[m+1];
    a[n]=b[n]='\0';
    printf("Enter the two sequences: \n");
    while((getchar())!='\n');
```

```
    fgets(a,n+1,stdin);
    while((getchar())!='\n');
    fgets(b,m+1,stdin);
    int arr[m+1][n+1];
    for(int i=0;i<=n;i++){
        arr[0][i]=0;
    }
    for(int i=0;i<=m;i++){
        arr[i][0]=0;
    }
    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(a[j-1]==b[i-1]){
                arr[i][j]=arr[i-1][j-1]+1;
            }
            else{
                arr[i][j]=__max(arr[i-1][j],arr[i][j-1]);
            }
        }
    }
    printf("Matrix of solutions: \n");
    printf("%c\t%3d ",'-',0);
    for(int i=0;i<n;i++){
        printf("%3c ",a[i]);
    }
    printf("\n\n");
    for(int i=0;i<=m;i++){
        if(i==0){
            printf("%d \t",0);
        }
        else
        printf("%c\t",b[i-1]);
        for(int j=0;j<=n;j++){
            printf("%3d ",arr[i][j]);
        }
        printf("\n");
    }
    printf("The length of the longest common subsequence
is : %d\n",arr[m][n]);
    printf("One of the valid longest subsequences is:\n");
    char s[arr[m][n]+1];
    s[arr[m][n]]='\0';
```

```
    int i=m,j=n;
    while(arr[i][j]>0){
        if(arr[i-1][j]==arr[i][j]){
            i--;
        }
        else{
            s[arr[i][j]-1]=b[i-1];
            i--;
            j--;
        }
    }
    if(arr[m][n]!=0)
    printf("%s\n",s);
    else
    printf("No common subsequence\n");

return 0;
}
```

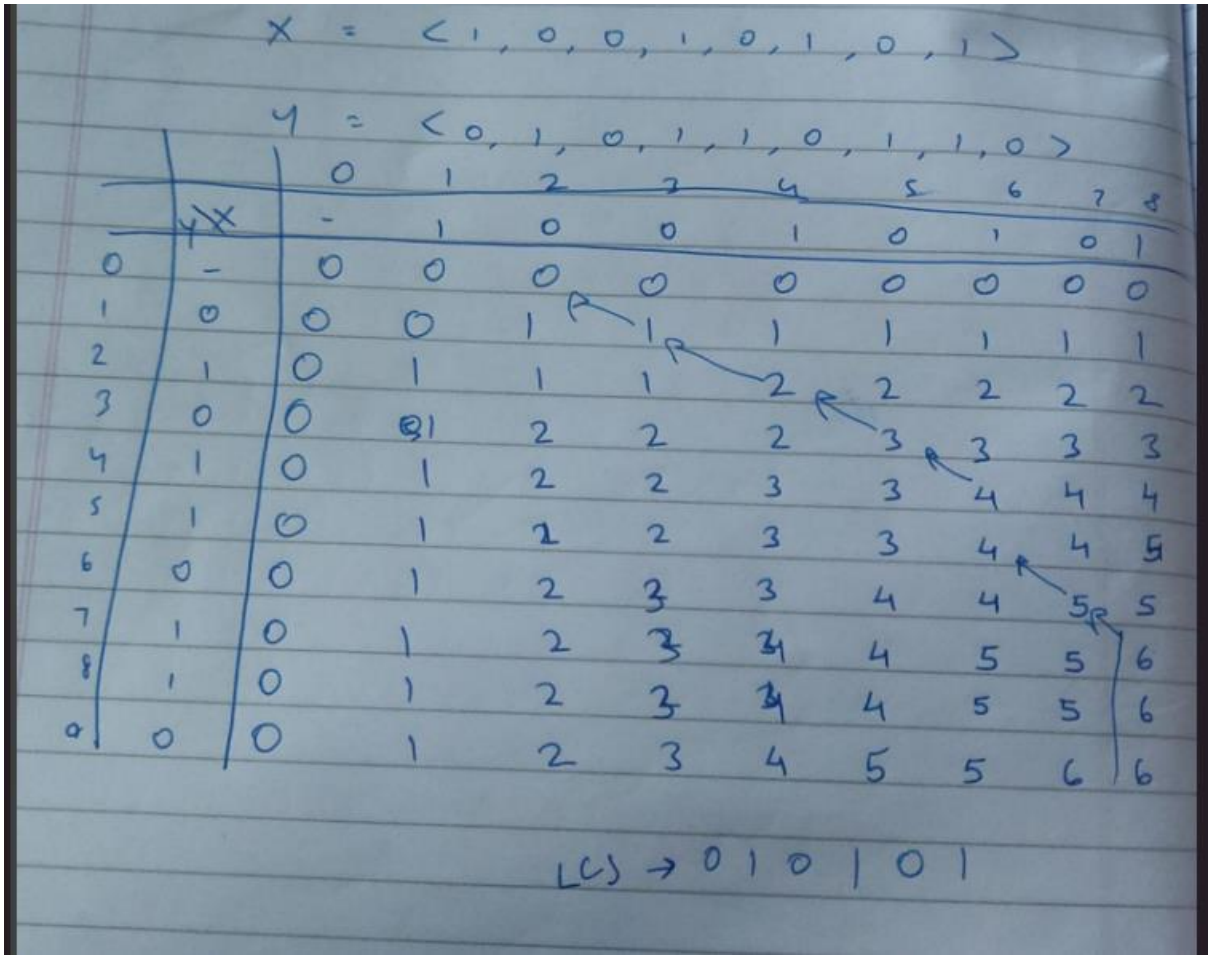
Output:



Conclusion:

- This solution has an optimal substructure i.e. the final solution contain solutions to all subproblems too, as it uses dynamic programming.
- The solution to this problem using the naïve approach of taking a subarray and checking whether it exists in the other string has O(2^m*n) time complexity, where m and n are the lengths of the two strings respectively. This solution using the dynamic programming approach only takes O(n*n) time.

Rough working:



Theory:

Dynamic Programming (DP) is a powerful technique used for solving optimization, search, and counting problems that can be decomposed into overlapping subproblems. Some features of Dynamic programming are:

1. **Overlapping Subproblems**: DP is used when a problem has overlapping subproblems. This means that the same subproblems are used to solve many different larger problems.
2. **Optimal Substructure**: A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.
3. **Memoization**: DP uses memoization to remember the results of expensive function calls and reuse them when the same inputs occur again.
4. **Bottom-Up Approach**: DP typically fills up an n-dimensional table. Iteratively, we solve the problem in a bottom-up manner, i.e., by solving all related sub-problems first, typically by filling up an n-dimensional table.

In the context of the **Assembly Line Scheduling problem**, dynamic programming is used to find the fastest way to travel through the assembly line. The problem has an optimal substructure as the fastest way to a station depends only on the fastest way to its preceding stations. The overlapping subproblems property is used as the fastest way to a station is calculated once and stored for future use.

In the **Longest Common Subsequence (LCS) problem**, dynamic programming is used to find the longest subsequence common to all sequences in a set of sequences. The problem has an optimal substructure as the LCS of the input sequences depends on the LCS of the suffixes. The overlapping subproblems property is used as the LCS of the suffixes is calculated once and stored for future use.