Name	Surti Keyur
Roll No.	25MCE026
Branch	M.Tech (CSE)

Experiment 7

Aim: Implement Assembly Line Scheduling problem using dynamic programming concepts.

Objective

- To understand the concept of Dynamic Programming.
- To learn how to optimize sequential decisions with overlapping subproblems.
- To find the minimum assembly time and optimal path through stations.

Problem Statement

A car chassis must pass through n stations on each of two assembly lines. Each station S[i][j] (for line i and station j) performs a specific operation taking a[i][j] time units. After each station, the chassis can either stay on the same line or switch to the other line (with a transfer time). The task is to find:
1. The minimum total assembly time.
2. The sequence of lines (path) followed to achieve this minimum time.

Algorithm (Dynamic Programming Approach)

1. Initialize starting times:
   f1[0] = entry1 + a1[0]
   f2[0] = entry2 + a2[0]

2. For each station j from 1 to n-1:
   f1[j] = min(f1[j-1] + a1[j], f2[j-1] + t21[j-1] + a1[j])
   f2[j] = min(f2[j-1] + a2[j], f1[j-1] + t12[j-1] + a2[j])
   Keep track of which line (1 or 2) was chosen.

3. Add exit times:
   f1[n] = f1[n-1] + exit1
   f2[n] = f2[n-1] + exit2

4. Choose the minimum:

result = min(f1[n], f2[n])

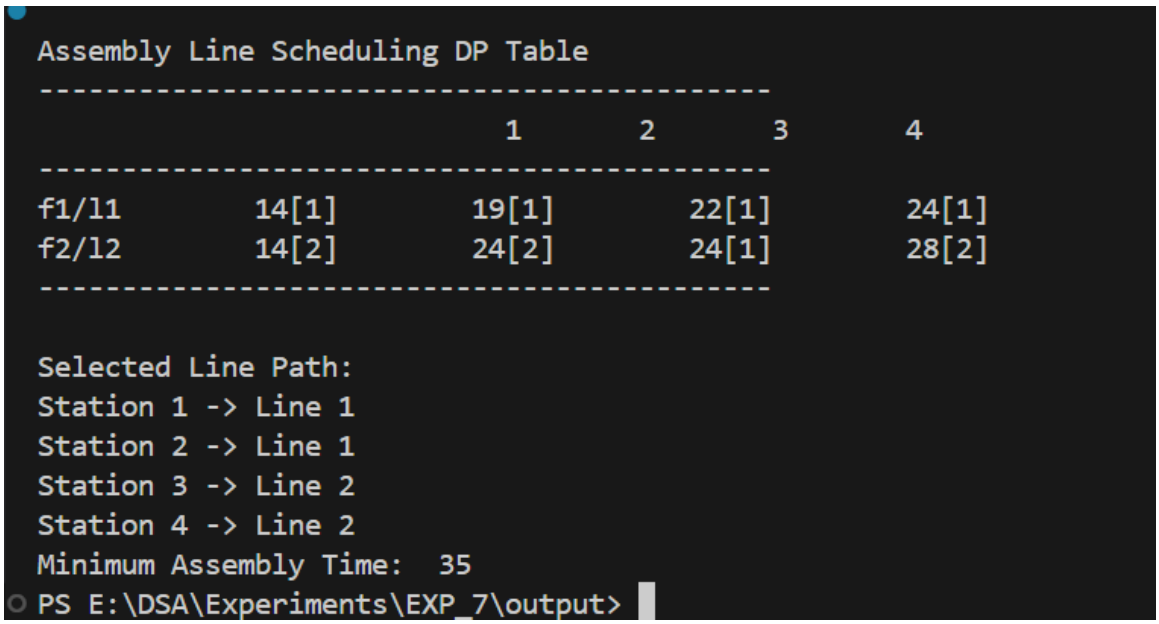
5. Backtrack to find the selected line path.

Source Code

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

int main() {
    const int numStations = 4;
    int entry[2] = {10, 12};
    int exitTime[2] = {18, 7};
    int a[2][numStations] = {{4, 5, 3, 2}, {2, 10, 1, 4}};
    int t[2][numStations - 1] = {{7, 4, 5}, {9, 2, 8}};

    int f1[numStations], f2[numStations], l1[numStations], l2[numStations];

    f1[0] = entry[0] + a[0][0];
    f2[0] = entry[1] + a[1][0];

    for (int j = 1; j < numStations; j++) {
        if (f1[j-1] + a[0][j] <= f2[j-1] + t[1][j-1] + a[0][j]) {
            f1[j] = f1[j-1] + a[0][j];
            l1[j] = 1;
        } else {
            f1[j] = f2[j-1] + t[1][j-1] + a[0][j];
            l1[j] = 2;
        }

        if (f2[j-1] + a[1][j] <= f1[j-1] + t[0][j-1] + a[1][j]) {
            f2[j] = f2[j-1] + a[1][j];
            l2[j] = 2;
        } else {
            f2[j] = f1[j-1] + t[0][j-1] + a[1][j];
            l2[j] = 1;
        }
    }

    int fFinal1 = f1[numStations - 1] + exitTime[0];
    int fFinal2 = f2[numStations - 1] + exitTime[1];

    int finalTime, lastLine;
    if (fFinal1 <= fFinal2) {
        finalTime = fFinal1;
        lastLine = 1;
    } else {
        finalTime = fFinal2;
        lastLine = 2;
```

```
    }

    cout << "\nAssembly Line Scheduling DP Table\n";
    cout << "-------------------------------------------\n";
    cout << "                    1       2       3       4\n";
    cout << "-------------------------------------------\n";
    cout << "f1/l1      ";
    for (int j = 0; j < numStations; j++)
        cout << setw(8) << f1[j] << "[" << l1[j] << "]";
    cout << "\n";
    cout << "f2/l2      ";
    for (int j = 0; j < numStations; j++)
        cout << setw(8) << f2[j] << "[" << l2[j] << "]";
    cout << "\n-------------------------------------------\n";

    int line[numStations];
    line[numStations - 1] = lastLine;
    for (int j = numStations - 1; j > 0; j--) {
        if (line[j] == 1)
            line[j - 1] = l1[j];
        else
            line[j - 1] = l2[j];
    }

    cout << "\nSelected Line Path:\n";
    for (int j = 0; j < numStations; j++)
        cout << "Station " << j + 1 << " -> Line " << line[j] << endl;

    cout << "Minimum Assembly Time:  " << finalTime << endl;
    return 0;
}
```

Output



Result / Conclusion

After executing the program, the minimum total assembly time obtained is 35 time units. The optimal path for processing the chassis through the stations is:
Station 1 → Line 1
Station 2 → Line 1
Station 3 → Line 2
Station 4 → Line 2.

This means the product should start on Line 1, continue on Line 1 for the first two stations, then switch to Line 2 at the third station, and stay on Line 2 until the end to achieve the minimum total assembly time.

This experiment demonstrates how Dynamic Programming (DP) effectively solves problems where decisions depend on previous results — where overlapping subproblems and optimal substructure exist. Instead of recomputing total time for every path, DP stores intermediate results and builds the final result step by step, avoiding redundant calculations and guaranteeing the optimal solution.

In real-world scenarios, this approach can optimize car manufacturing, semiconductor production, and workflow scheduling, leading to better efficiency, lower production time, and cost savings.

Complexity Analysis

1. Time Complexity:

For n stations, at each step we compute the time for both lines based on the previous station's results. Each computation takes constant time, so total time complexity is **O(n)**. In comparison, a brute-force approach explores all 2^n possible paths, which is exponential. Thus, Dynamic Programming reduces exponential complexity to linear.

2. Space Complexity:

We use arrays to store minimum times and path decisions for each station (f1[], f2[], l1[], l2[]). Each array is of length n, so space complexity is **O(n)**. If path reconstruction is not needed, we could reduce space further to O(1) by storing only the last computed values.