

# UNIT-II

# Dynamic Programming :

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Mostly, dynamic programming algorithms are used for solving optimization problems. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best optimal final solution. This paradigm is thus said to be using Bottom-up approach.

**Dynamic programming can be used in both top-down and bottom-up manner.**

- So we can conclude that –
- The problem should be able to be divided into smaller overlapping sub-problem.
- Final optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use memorization.

To design an algorithm for a problem using Dynamic Programming, the problem we want to solve must have these two properties:

**Overlapping Subproblems:** Means that the problem can be broken down into smaller subproblems, where the solutions to the subproblems are overlapping. Having subproblems that are overlapping means that the solution to one subproblem is part of the solution to another subproblem.

**Optimal Substructure:** Means that the complete solution to a problem can be constructed from the solutions of its smaller subproblems. So not only must the problem have overlapping subproblems, the substructure must also be optimal so that there is a way to piece the solutions to the subproblems together to form the complete solution.

# How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memorization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

# Approaches of dynamic programming :

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

## ► **Top-down approach :**

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

## ► **Bottom-Up approach :**


The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

## Factorial in Maths :

The factorial of a natural number  $n$  indicates the number of ways  $n$  items can be arranged. It plays an important role in various mathematical concepts such as permutations, combinations, probability, and many others. For a positive integer  $n$ , the value of factorial is multiplication of all positive integers less than or equal to  $n$ .

**Factorial of  $n = n \times (n-1) \times (n-2) \times \dots \times 1$**

If there are three distinct objects,  
in how many ways you can arrange them ?



01		04	
02		05	
03		06	

You can arrange them in six different ways,  
which can be expressed as

$$3! = 3 \times 2 \times 1 = 6$$



The notation of the factorial function is "!" or "J". If we have to find the factorial of the number n then, it is written as n! or nJ. Let's understand it with some examples:

$0! = 1$  ( Value of Factorial 0 is 1 because it shows the number of possible ways to arrange a data set with no value in it is 1)

$$1! = 1$$

$$2! = 2 \times 1 = 2$$

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

## Factorial 1 to 15

1! = 1	6! = 720	11! = 39,916,800
2! = 2	7! = 5040	12! = 479,001,600
3! = 6	8! = 40320	13! = 6,227,020,800
4! = 24	9! = 362880	14! = 87,178,291,200
5! = 120	10! = 3628800	15! = 1,307,674,368,000

## Coefficient of a Variable :

The coefficient of a variable is a number that is multiplied by a variable in an algebraic expression. It's often a constant that represents the number of units of the variable.

Coefficient in Math is a number or alphabet or a symbol multiplied by a variable in an algebraic expression, showing the variable's impact on the expression. For example, in the expression  $4xy$ , the coefficient is 4 which is multiplied by the variable  $xy$ .



### Coefficient of a Variable

$$5x^2 + 8y + 2$$

Coefficients

Constant



# Binomial Theorem :

Binomial theorem is a fundamental principle in algebra that describes the algebraic expansion of powers of a binomial. According to this theorem, the expression  $(a + b)^n$  where  $a$  and  $b$  are any numbers and  $n$  is a non-negative integer. It can be expanded into the sum of terms involving powers of  $a$  and  $b$ .

Binomial theorem is used to find the expansion of two terms hence it is called the Binomial Theorem.

## Binomial Theorem



$$(a + b)^0 = 1$$

$$(a + b)^1 = a + b$$

$$(a + b)^2 = a^2 + 2ab + b^2$$

$$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

## Binomial Theorem Statement :

Binomial theorem for the expansion of  $(a+b)^n$  is stated as,

$$(a + b)^n = {}^nC_0 a^n b^0 + {}^nC_1 a^{n-1} b^1 + {}^nC_2 a^{n-2} b^2 + \dots + {}^nC_r a^{n-r} b^r + \dots + {}^nC_n a^0 b^n$$

where  $n > 0$  and the  ${}^nC_k$  is the binomial coefficient.

## Binomial Coefficients

Binomial Coefficients are positive integers represented as  ${}^nC_k$  where  $n \geq k \geq 0$ . The following are important properties of Binomial Coefficients.

The value of  ${}^nC_k$  represents the number of possibilities to pick "k" items from n elements.

The formula for  ${}^nC_k$  is  $n! / \{k! (n - k)!\}$  where ! represents factorial.

These binomial coefficients of the Binomial Theorem, which gives a formula for expanding statements such as

$$(a + b)^n = {}^nC_0 a^n b^0 + {}^nC_1 a^{n-1} b^1 + {}^nC_2 a^{n-2} b^2 + \dots + {}^nC_r a^{n-r} b^r + \dots + {}^nC_n a^0 b^n.$$

For instance, the binomial coefficients in  $(x + y)^3$  are 1, 3, 3, and 1.

# How to Calculate Binomial Coefficients

Using Factorial Notation. The binomial coefficient  $\binom{n}{r}$  can be calculated using factorials:

$$\binom{n}{r} = n! / r! (n - r)!$$

Where  $n!$  ( $n$  factorial) is the product of all positive integers up to  $n$ .

For example:

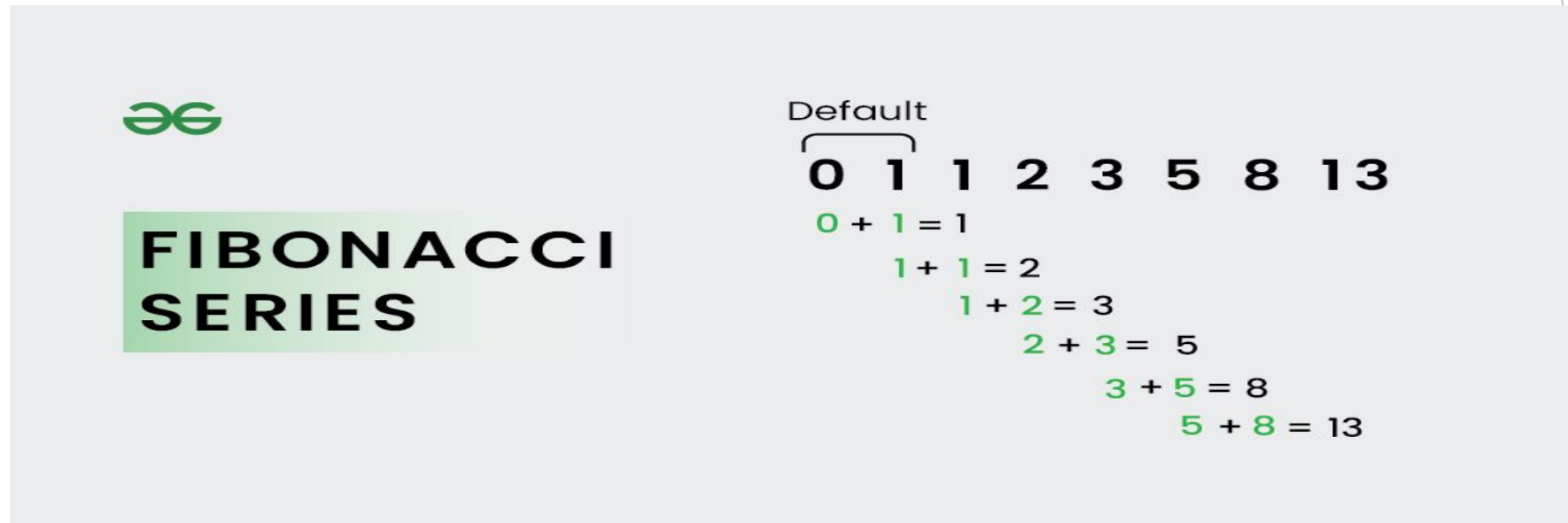
$$\begin{aligned}\binom{5}{2} &= 5! / (2 \times 1) (3 \times 2 \times 1) \\ &= (5 \times 4 \times 3 \times 2 \times 1) / (2 \times 1 \times 3 \times 2 \times 1) \\ &= (5 \times 4) / (2 \times 1) = 10\end{aligned}$$

```
// Binomial Coefficient using recursion
#include <bits/stdc++.h>
using namespace std;
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    // n can not be greater than k so we return 0 here
    if (k > n)
        return 0;
    // base condition when k and n are equal or k = 0
    if (k == 0 || k == n)
        return 1;
    // Recursive add the value
    return binomialCoeff(n - 1, k - 1) + binomialCoeff(n - 1, k);
}
```

```
int main()
{
    int n = 5, k = 2;
    cout << binomialCoeff(n, k);
    return 0;
}
```

# What is the Fibonacci Series?

The Fibonacci series is the sequence where each number is the sum of the previous two numbers of the sequence. The first two numbers of the Fibonacci series are 0 and 1 and are used to generate the Fibonacci series.



How to Find the Nth term of Fibonacci Series?

In mathematical terms, the number at the nth position can be represented by:

$$F_n = F_{n-1} + F_{n-2}$$

where,  $F_0 = 0$  and  $F_1 = 1$ .

For example, Fibonacci series 10 terms are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34



## Key points

- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

## Let's understand through an example.

Suppose we have an array that has 0 and 1 values at  $a[0]$  and  $a[1]$  positions, respectively shown as below:



Since the bottom-up approach starts from the lower values, so the values at  $a[0]$  and  $a[1]$  are added to find the value of  $a[2]$  shown as below:

0	1	1	
$a[0]$	$a[1]$	$a[2]$	

The value of  $a[3]$  will be calculated by adding  $a[1]$  and  $a[2]$ , and it becomes 2 shown as below:

0	1	1	2	
$a[0]$	$a[1]$	$a[2]$	$a[3]$	

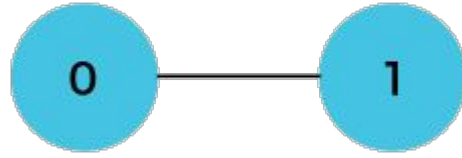
The value of  $a[4]$  will be calculated by adding  $a[2]$  and  $a[3]$ , and it becomes 3 shown as below:

0	1	1	2	3	
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	

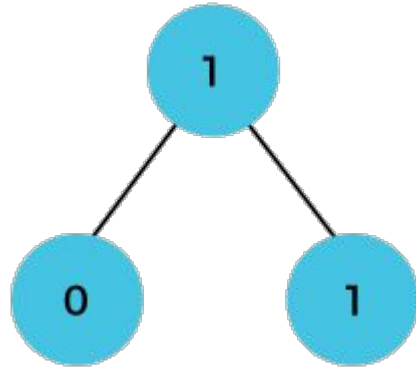
The value of  $a[5]$  will be calculated by adding the values of  $a[4]$  and  $a[3]$ , and it becomes 5 shown as below:

0	1	1	2	3	5	
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	

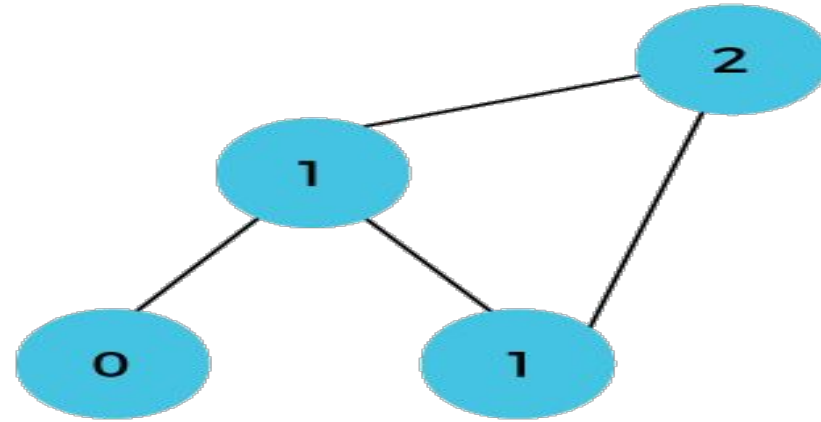
**Let's understand through the diagrammatic representation.**  
Initially, the first two values, i.e., 0 and 1 can be represented as:



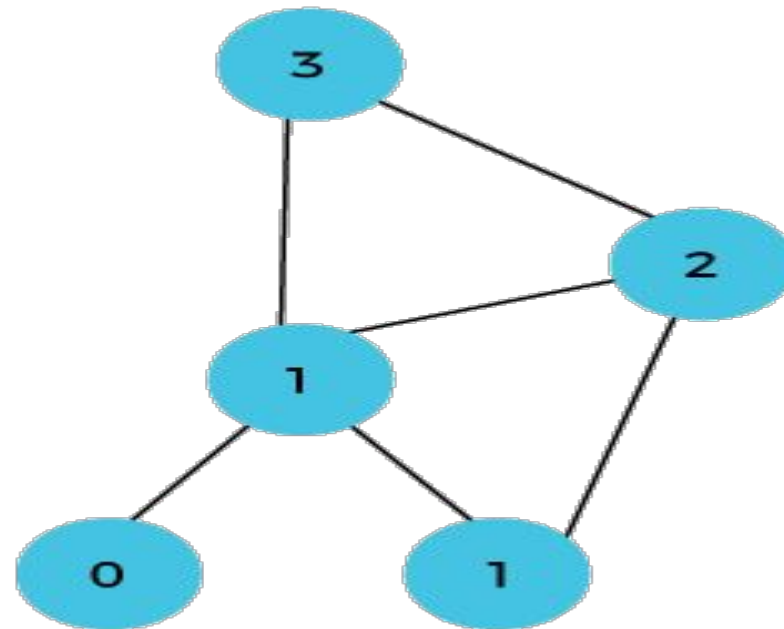
When  $i=2$  then the values 0 and 1 are added shown as below:



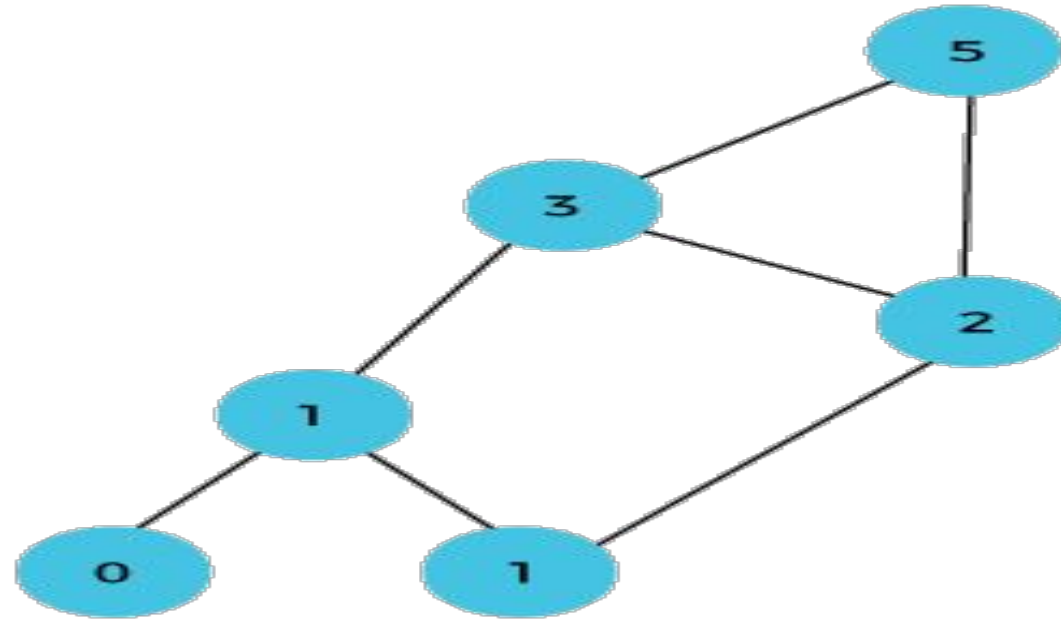
When  $i=3$  then the values 1 and 1 are added shown as below:



When  $i=4$  then the values 2 and 1 are added shown as below:



When  $i=5$ , then the values 3 and 2 are added shown as below:





```
// C++ Program to print the fibonacci series using iteration (loops)
#include <iostream>
using namespace std;
// Function to print fibonacci series
void printFib(int n)
{
    if (n < 1)
    {
        cout << "Invalid Number of terms\n";
        return;
    }
    // When number of terms is greater than 0
    int prev1 = 1;
    int prev2 = 0;
    cout << prev2 << " ";
```

```
// If n is 1, then we do not need to proceed further
if (n == 1)
    return;
cout << prev1 << " ";
// Print 3rd number onwards using
// the recursive formula
for (int i = 3; i <= n; i++)
{
    int curr = prev1 + prev2;
    prev2 = prev1;
    prev1 = curr;
    cout << curr << " ";
}
}
```



```
// Driver code
```

```
int main()
```

```
{
```

```
    int n = 9;
```

```
    printFib(n);
```

```
    return 0;
```

```
}
```

## Output

0 1 1 2 3 5 8 13 21

## Complexity Analysis

**Time Complexity:**  $O(n)$

**Auxiliary Space:**  $O(1)$

## **Naive Approach] Using Recursion:**

We can use recursion to solve this problem because any Fibonacci number  $n$  depends on previous two Fibonacci numbers. Therefore, this approach repeatedly breaks down the problem until it reaches the base cases.

### **Recurrence relation:**

Base case:  $F(n) = n$ , when  $n = 0$  or  $n = 1$

Recursive case:  $F(n) = F(n-1) + F(n-2)$  for  $n > 1$

## ► **Graph :**

- A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -
  - **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of  $V$  vertices and a set of  $E$  edges, each edge connecting two different vertices.
  - **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
  - **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

# Floyd Warshall Algorithm

The **Floyd Warshall Algorithm** is an all-pair shortest path algorithm that uses Dynamic Programming to find the shortest distances between every pair of vertices in a graph, unlike Dijkstra and Bellman-Ford which are single source shortest path algorithms. This algorithm works for both the **directed** and **undirected weighted** graphs and can handle graphs with both **positive** and **negative weight edges**.

**Note:** It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved using the Adjacency Matrix representation of graphs.

## Floyd-Warshall Algorithm

Consider a graph,  $G = \{V, E\}$  where  $V$  is the set of all vertices present in the graph and  $E$  is the set of all the edges in the graph. The graph,  $G$ , is represented in the form of an adjacency matrix,  $A$ , that contains all the weights of every edge connecting two vertices.



# Algorithm

**Step 1** – Construct an adjacency matrix  $A$  with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as  $\infty$ .

**Step 2** – Derive another adjacency matrix  $A_1$  from  $A$  keeping the first row and first column of the original adjacency matrix intact in  $A_1$ . And for the remaining values, say  $A_1[i,j]$ , if  $A[i,j] > A[i,k] + A[k,j]$  then replace  $A_1[i,j]$  with  $A[i,k] + A[k,j]$ . Otherwise, do not change the values. Here, in this step,  $k = 1$  (first vertex acting as pivot).

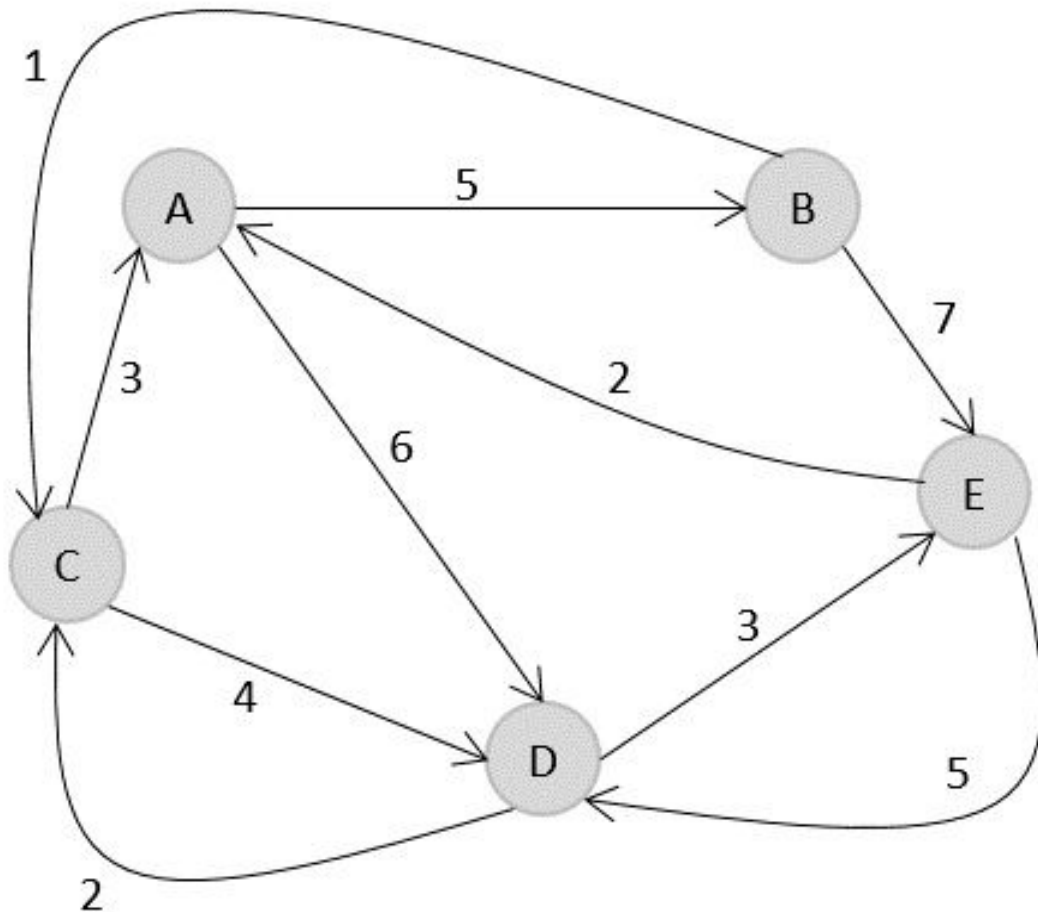
**Step 3** – Repeat Step 2 for all the vertices in the graph by changing the  $k$  value for every pivot vertex until the final matrix is achieved.

**Step 4** – The final adjacency matrix obtained is the final solution with all the shortest paths.

```
Floyd-Warshall(w, n){ // w: weights, n: number of vertices
  for i = 1 to n do // initialize, D (0) = [wij]
    for j = 1 to n do{
      d[i, j] = w[i, j];
    }
  for k = 1 to n do // Compute D (k) from D (k-1)
    for i = 1 to n do
      for j = 1 to n do
        if (d[i, k] + d[k, j] < d[i, j]){
          d[i, j] = d[i, k] + d[k, j];
        }
      }
    return d[1..n, 1..n];
}
```

# Example

Consider the following directed weighted graph  $G = \{V, E\}$ . Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



### Step 1

Construct an adjacency matrix **A** with all the distances as values.

$$A = \begin{matrix} & \begin{matrix} 0 & 5 & \infty & 6 & \infty \end{matrix} \\ \begin{matrix} \infty \\ 3 \\ \infty \\ 2 \end{matrix} & \begin{matrix} 0 \\ \infty \\ 0 \\ \infty \end{matrix} & \begin{matrix} 1 \\ \infty \\ 2 \\ \infty \end{matrix} & \begin{matrix} \infty \\ 4 \\ 0 \\ 5 \end{matrix} & \begin{matrix} 7 \\ \infty \\ 3 \\ 0 \end{matrix} \end{matrix}$$

### Step 2

Considering the above adjacency matrix as the input, derive another matrix  $A_0$  by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A = \begin{matrix} & \begin{matrix} 0 & 5 & \infty & 6 & \infty \end{matrix} \\ \begin{matrix} \infty \\ 3 \\ \infty \\ 2 \end{matrix} & \begin{matrix} 0 \\ \infty \\ 0 \\ \infty \end{matrix} & \begin{matrix} 1 \\ \infty \\ 2 \\ \infty \end{matrix} & \begin{matrix} \infty \\ 4 \\ 0 \\ 5 \end{matrix} & \begin{matrix} 7 \\ \infty \\ 3 \\ 0 \end{matrix} \end{matrix}$$
$$A_1 = \begin{matrix} & \begin{matrix} 0 & 5 & \infty & 6 & \infty \end{matrix} \\ \begin{matrix} \infty \\ 3 \\ \infty \\ 2 \end{matrix} & \begin{matrix} 0 \\ 8 \\ \infty \\ 7 \end{matrix} & \begin{matrix} 1 \\ \infty \\ 2 \\ \infty \end{matrix} & \begin{matrix} \infty \\ 4 \\ 0 \\ 5 \end{matrix} & \begin{matrix} 7 \\ \infty \\ 3 \\ 0 \end{matrix} \end{matrix}$$

### Step 3

Considering the above adjacency matrix as the input, derive another matrix  $A_0$  by keeping only first rows and columns intact. Take  $k = 1$ , and replace all the other values by  $A[i,k] + A[k,j]$ .

$$A_2 = \begin{matrix} & & 5 & & & \\ & \infty & 0 & 1 & \infty & 7 \\ & 8 & & & & \\ & \infty & & & & \\ & 7 & & & & \\ & 0 & 5 & 6 & 6 & 12 \\ & \infty & 0 & 1 & \infty & 7 \\ A_2 = & 3 & 8 & 0 & 4 & 15 \\ & \infty & \infty & 2 & 0 & 3 \\ & 2 & 7 & 8 & 5 & 0 \end{matrix}$$

# Step 4

Considering the above adjacency matrix as the input, derive another matrix **A<sub>0</sub>** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

6

1

3

8

0

4

15

2

8

0

5

6

6

12

4

0

1

5

7

3

8

0

4

15

5

10

2

0

3

2

7

8

5

0





## Step 5

Considering the above adjacency matrix as the input, derive another matrix  $\mathbf{A_0}$  by keeping only first rows and columns intact. Take  $\mathbf{k = 1}$ , and replace all the other values by  $\mathbf{A[i,k]+A[k,j]}$ .

$$A_4 = \begin{matrix} & & & 6 & & \\ & & & 5 & & \\ & & & 4 & & \\ 5 & 10 & 2 & 0 & 3 & \\ & & & 5 & & \\ & 0 & 5 & 6 & 6 & 9 \\ & 4 & 0 & 1 & 5 & 7 \\ A_4 = & 3 & 8 & 0 & 4 & 7 \\ & 5 & 10 & 2 & 0 & 3 \\ & 2 & 7 & 7 & 5 & 0 \end{matrix}$$

## Step 6

Considering the above adjacency matrix as the input, derive another matrix **A<sub>0</sub>** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_5 = \begin{matrix} & & & & 9 \\ & & & & 7 \\ & & & & 7 \\ & & & & 3 \\ & 2 & 7 & 7 & 5 & 0 \\ 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ A_5 = 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

## Analysis

From the pseudocode above, the Floyd-Warshall algorithm operates using three for loops to find the shortest distance between all pairs of vertices within a graph.

Therefore, the **time complexity** of the Floyd-Warshall algorithm is **O(n<sup>3</sup>)**, where 'n' is the number of vertices in the graph. The **space complexity** of the algorithm is **O(n<sup>2</sup>)**.

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;
// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(vector<vector<int>> &graph) {
    int V = graph.size();
    // Add all vertices one by one to
    // the set of intermediate vertices.
    for (int k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++) {
```

```
// Pick all vertices as destination
// for the above picked source
for (int j = 0; j < V; j++) {

    // If vertex k is on the shortest path from
    // i to j, then update the value of graph[i][j]

    if ((graph[i][j] == -1 ||
        graph[i][j] > (graph[i][k] + graph[k][j]))
        && (graph[k][j] != -1 && graph[i][k] != -1))
        graph[i][j] = graph[i][k] + graph[k][j];
    }
}
}
```

```
int main() {  
    vector<vector<int>> graph = {  
        {0, 4, -1, 5, -1},  
        {-1, 0, 1, -1, 6},  
        {2, -1, 0, 3, -1},  
        {-1, -1, 1, 0, 2},  
        {1, -1, -1, 4, 0}  
    };  
    floydWarshall(graph);  
    for(int i = 0; i<graph.size(); i++) {  
        for(int j = 0; j<graph.size(); j++) {  
            cout<<graph[i][j]<<" ";  
        }  
        cout<<endl;  
    }  
    return 0;  
}
```

## Assembly Line Scheduling :

Assembly line scheduling is a manufacturing problem. In automobile industries assembly lines are used to transfer parts from one station to another station.

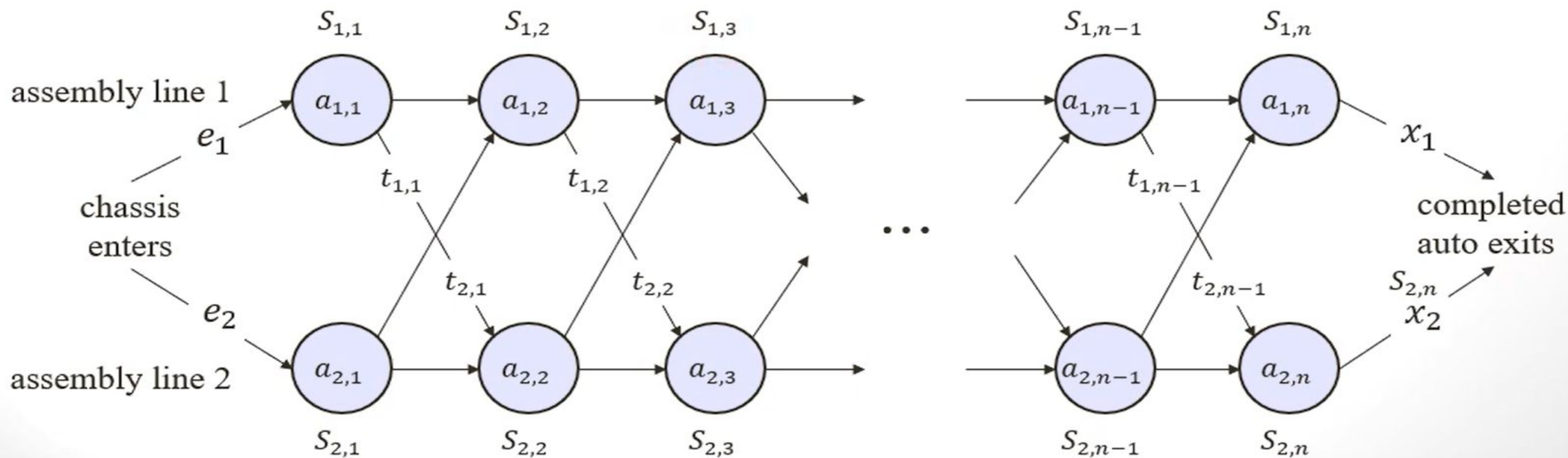
- Manufacturing of large items like car, trucks etc. generally undergoes through multiple stations, where each station is responsible for assembling particular part only. Entire product be ready after it goes through predefined  $n$  stations in sequence.
- Manufacturing of car may be done through several stages like engine fitting, coloring, light fitting, fixing of controlling system, gates, seats and many other things.
- The particular task is carried out at the station dedicated to that task only. Based on the requirement there may be more than one assembly line.
- In case of two assembly lines if the load at station  $j$  at assembly 1 is very high, then components are transfer to station of assembly line 2 the converse is also true. This technique helps to speed ups the manufacturing process.
- The time to transfer partial product from one station to next station on the same assembly line is negligible. During rush factory may transfer partially completed auto from one assembly line to another, complete the manufacturing as quickly as possible

Assembly line scheduling is a problem in operations management that involves determining the optimal sequence of tasks or operations on an assembly line to minimize production costs or maximize efficiency. This problem can be solved using various data structures and algorithms. One common approach is dynamic programming, which involves breaking the problem down into smaller sub-problems and solving them recursively.



# Dynamic Programming: Assembly Line Scheduling

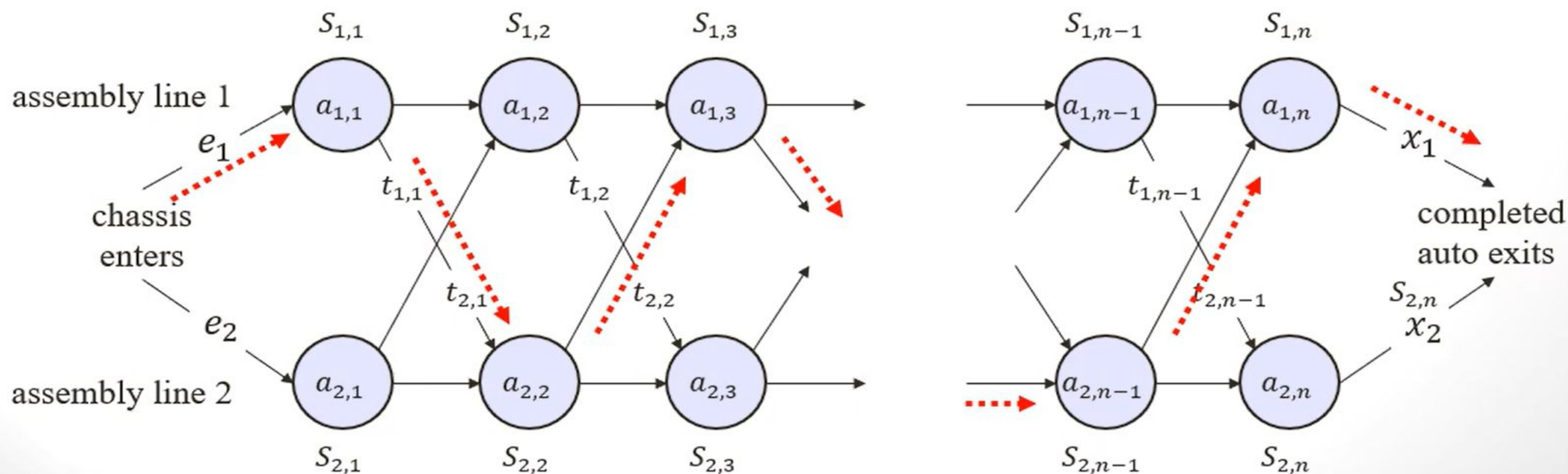
- Automobile factory has two assembly lines
  - Each line has  $n$  stations:  $S_{1,1}, \dots, S_{1,n}$  and  $S_{2,1}, \dots, S_{2,n}$ .
  - Corresponding stations  $S_{1,j}$  and  $S_{2,j}$  perform the same function but can take different amounts of time  $a_{1,j}$  and  $a_{2,j}$ .
  - Entry times are  $e_1$  and  $e_2$  and exit times are  $x_1$  and  $x_2$ .





# Dynamic Programming: Assembly Line Scheduling

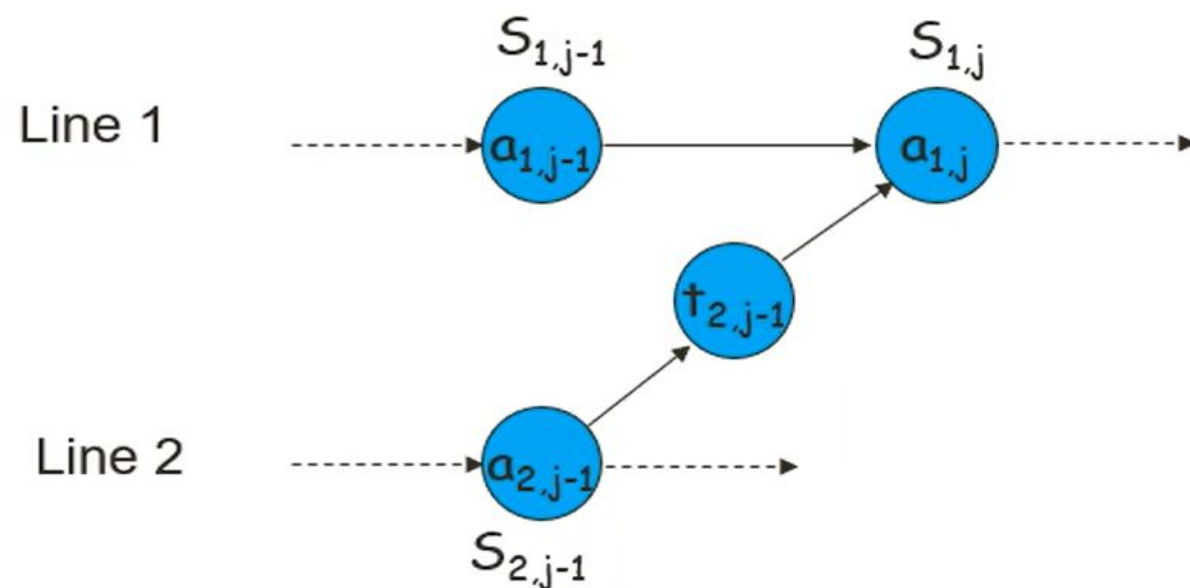
- After going through a station, car can either:
  - Stay on same line at no cost, or
  - Transfer to other line: cost after  $S_{i,j}$  is  $t_{i,j}, j = 1, \dots, n - 1$
- Problem:** what stations should be chosen from line 1 and line 2 in order to **minimize the total time through the factory for one car?**



# Dynamic Programming: Assembly Line Scheduling

## 1. Structure of the Optimal Solution

- How do we compute the minimum time of going through a station?
- Let's consider all possible ways to get from the starting point through station  $S_{1,j}$ 
  - We have two choices of how to get to  $S_{1,j}$ 
    - Through  $S_{1,j-1}$ , then directly to  $S_{1,j}$
    - Through  $S_{2,j-1}$ , then transfer over to  $S_{1,j}$



# Dynamic Programming: Assembly Line Scheduling

- **Generalization:** an optimal solution to the problem “*find the fastest way through  $S_{1,j}$* ” contains within it an optimal solution to subproblems: “*find the fastest way through  $S_{1,j-1}$  or  $S_{2,j-1}$* ”.
- This is referred to as the **optimal substructure** property
- We use this property to construct an optimal solution to a problem from optimal solutions to subproblems



# Dynamic Programming: Assembly Line Scheduling

## 2. Recursively define the value of an optimal solution

- We define the value of an optimal solution in terms of the optimal solution to subproblems
- **Definitions:**
  - $f^*$  : the fastest time to get through the entire factory
  - $f_i[j]$  : the fastest time to get from the starting point through station  $S_{i,j}$
  - $l^*$  : the line number which is used to exit the factory from the  $n^{th}$  station
  - $l_i[j]$  : the line number (1 or 2) whose  $S_{i,j-1}$  is used to reach  $S_{i,j}$ .

$$f^* = \min (f_1[n] + x_1, f_2[n] + x_2) \rightarrow \text{Objective Function}$$

## 2. Recursively define the value of an optimal solution

- **Base case:**  $j = 1, i = 1, 2$  (getting through station 1)

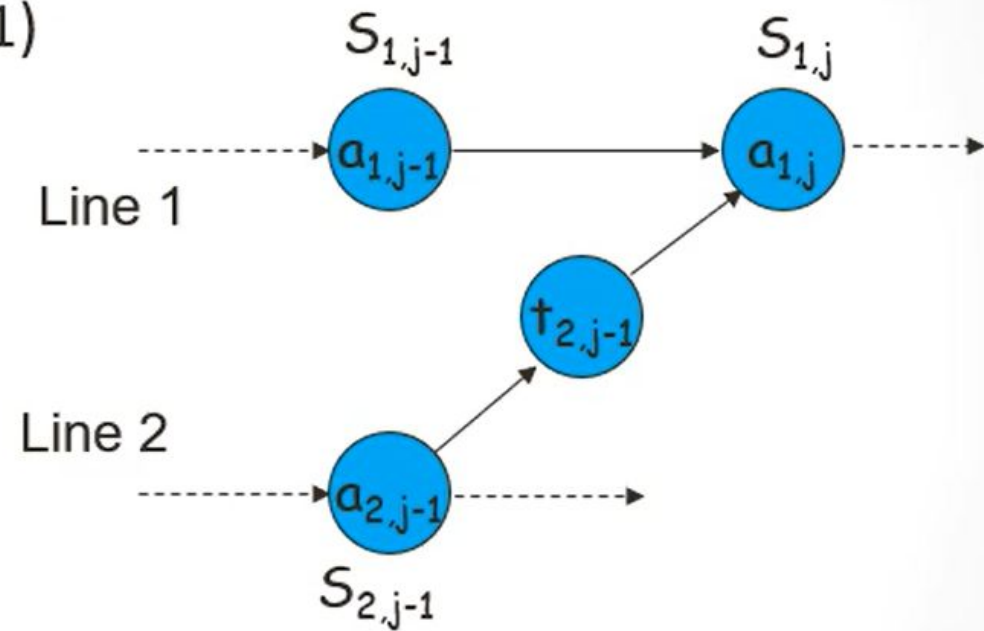
- $f_1[1] = e_1 + a_{1,1}$
- $f_2[1] = e_2 + a_{2,1}$

- **General case:**  $j = 2, 3, \dots, n$ , and  $i = 1, 2$

- The fastest way through  $S_{1,j}$  is either:

- The way through  $S_{1,j-1}$  then directly through  $S_{1,j}$  or  $f_1[j-1] + a_{1,j}$
- The way through  $S_{2,j-1}$ , transfer from line 2 to line 1, then through  $S_{1,j}$  or  $f_2[j-1] + t_{2,j-1} + a_{1,j}$

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$$



## 2. Recursively define the value of an optimal solution


$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

### 3. Compute the Optimal Solution

- Using bottom-up approach, first find optimal solutions to subproblems, and then use them to find an optimal solution to the problem.
- For  $j \geq 2$ , each value  $f_i[j]$  depends only on the values of  $f_1[j - 1]$  and  $f_2[j - 1]$
- Idea:** compute the values of  $f_i[j]$  as follows:

in increasing order of  $j$



	1	2	3	4	5
$f_1[j]$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$
$f_2[j]$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$

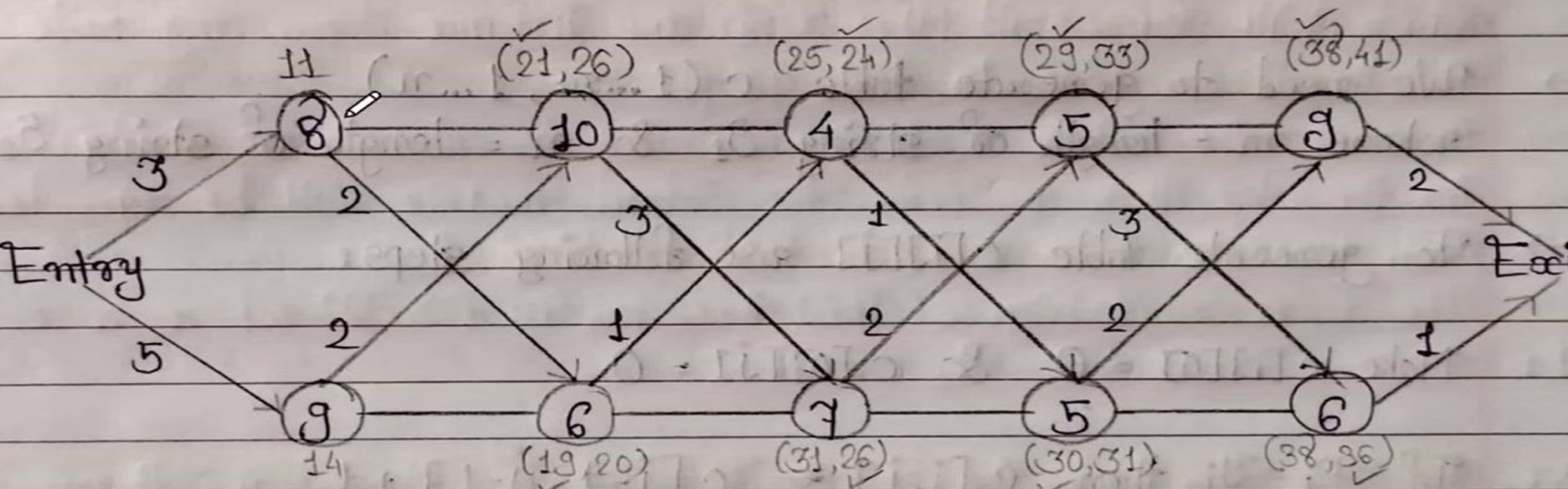


### 3. Explain Assembly Line-Scheduling with Example.

- Each line has  $n$  stations:  $S_{1,1}, \dots, S_{1,n}$  &  $S_{2,1}, \dots, S_{2,n}$ .
- Corresponding stations  $S_{1,j}$  &  $S_{2,j}$  perform the same function but can take different amounts of time  $a_{1,j}$  &  $a_{2,j}$ .
- Entry times are:  $e_1$  &  $e_2$ ; exit times are:  $x_1$  &  $x_2$ .
- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after  $S_{i,j}$  is  $t_{ij}$ ,  $j=1, \dots, n-1$ .

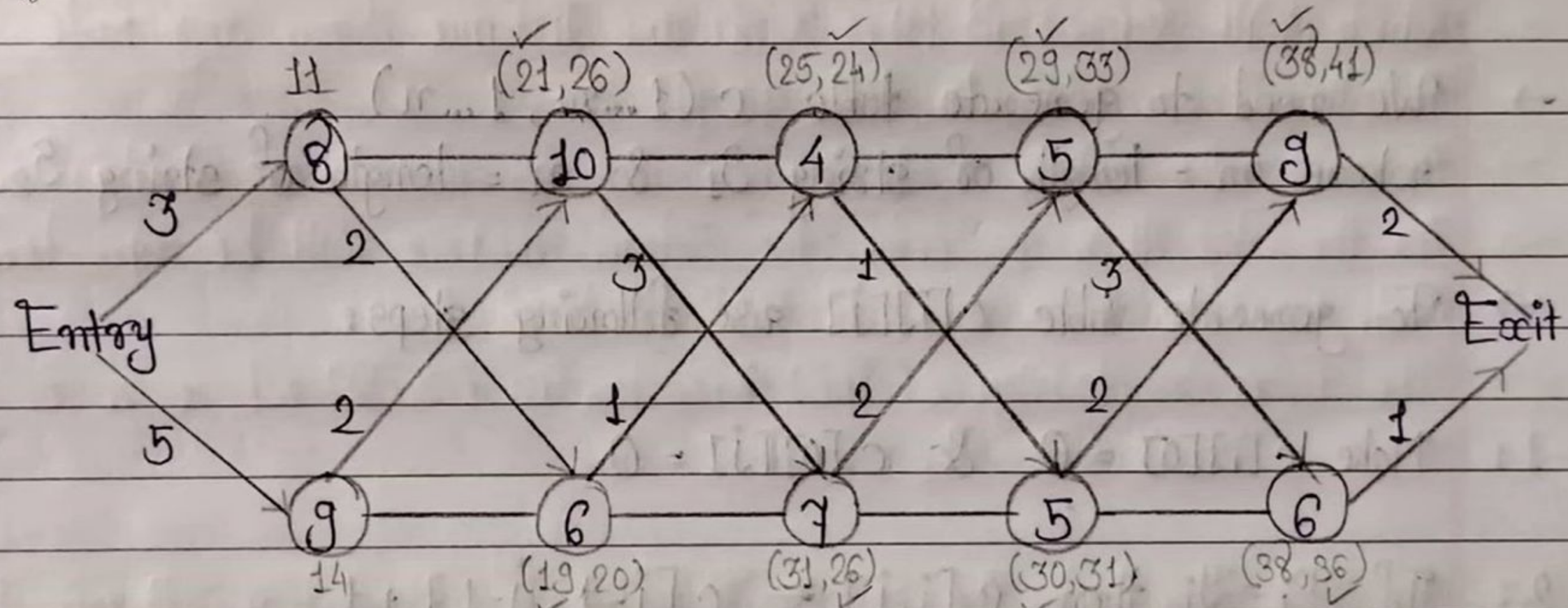


Stay on same line at no cost, or  
transfer to other line: Cost after  $g_{i,j}$  is  $t_{i,j}$ ,  $j=1, \dots, n-1$



j	1	2	3	4	5
$f_1[j]$	11	21	24	29	38
$f_2[j]$	14	19	26	30	36

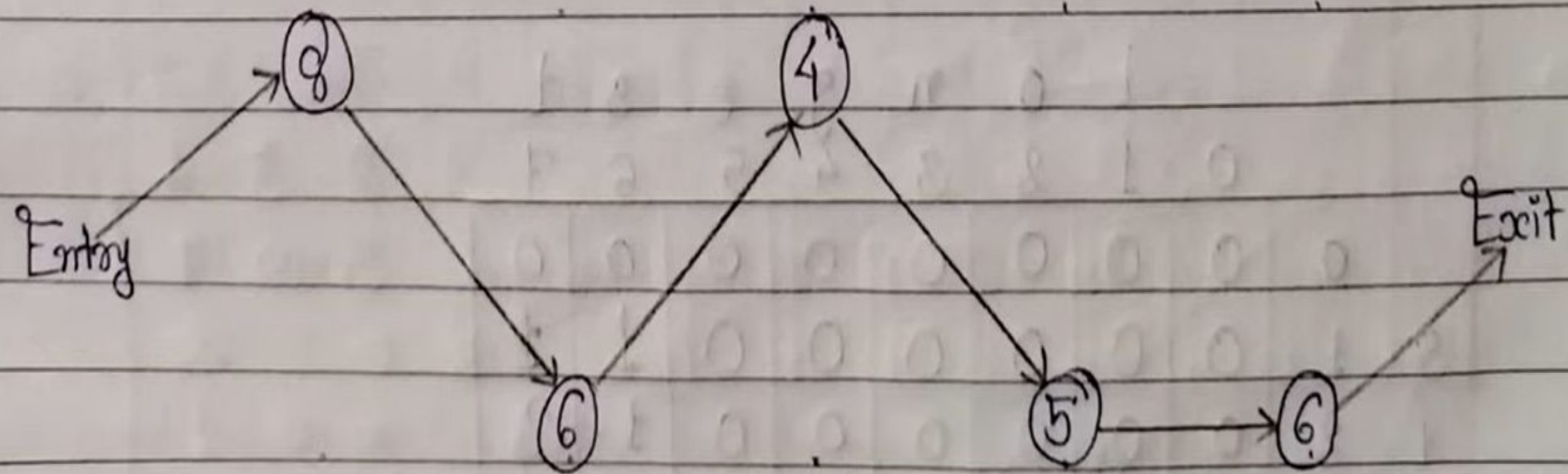
j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2



j	1	2	3	4	5
$f_1[j]$	11	21	24	29	38
$f_2[j]$	14	19	26	30	36

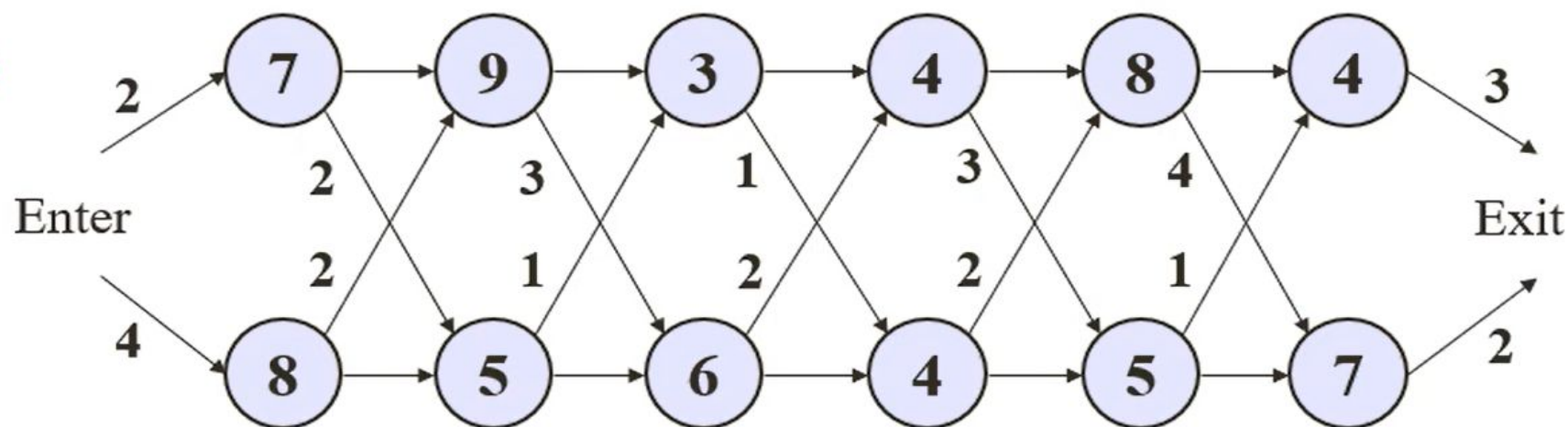
j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2





# Dynamic Programming: Assembly Line Scheduling

**Example:**



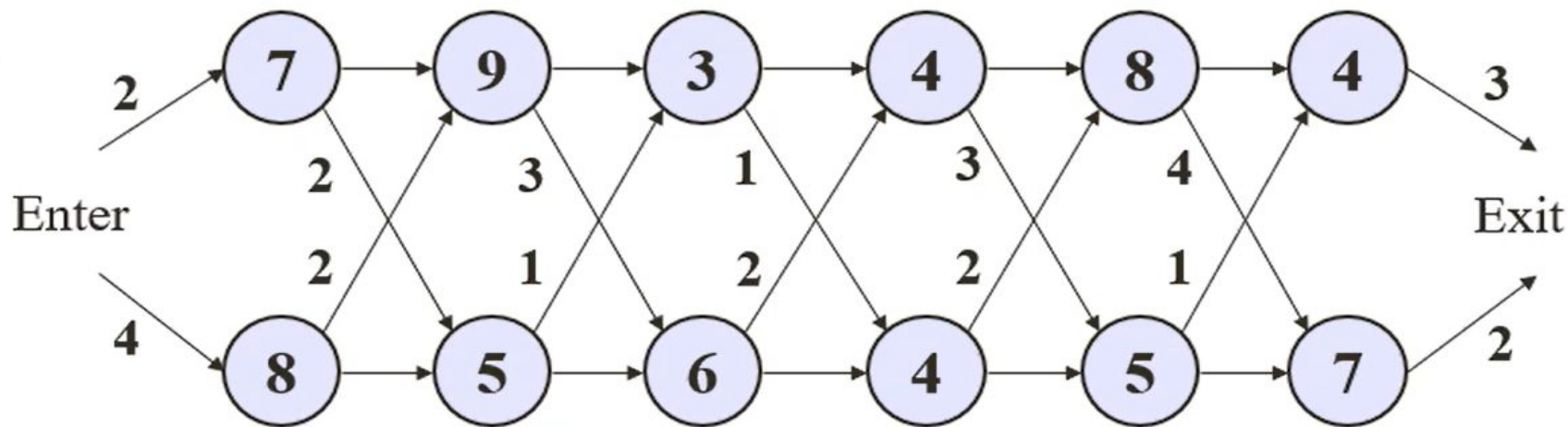
$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

	1	2	3	4	5	6
$f_1[j]$						
$f_2[j]$						

	1	2	3	4	5	6
$l_1[j]$						
$l_2[j]$						

# Dynamic Programming: Assembly Line Scheduling

**Example:**



$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f^* = 38$$

$$l^* = 1$$

	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

	1	2	3	4	5	6
$l_1[j]$	1	1	2	1	1	2
$l_2[j]$	2	1	2	1	2	2



# Dynamic Programming: Assembly Line Scheduling

FASTEST-WAY( $a, t, e, x, n$ )

```
1   $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2   $f_2[1] \leftarrow e_2 + a_{2,1}$ 
3  for  $j \leftarrow 2$  to  $n$ 
4      do if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5          then  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6               $l_1[j] \leftarrow 1$ 
7          else  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8               $l_1[j] \leftarrow 2$ 
9      if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10         then  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11              $l_2[j] \leftarrow 2$ 
12         else  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13              $l_2[j] \leftarrow 1$ 
14     if  $f_1[n] + x_1 \leq f_2[n] + x_2$ 
15         then  $f^* = f_1[n] + x_1$ 
16              $l^* = 1$ 
17     else  $f^* = f_2[n] + x_2$ 
18          $l^* = 2$ 
```

Compute initial values of  $f_1$  and  $f_2$

Time Complexity =  $O(n)$

Compute the values of  $f_1[j]$  and  $l_1[j]$

Compute the values of  $f_2[j]$  and  $l_2[j]$

Compute the values of the fastest time through the entire factory.

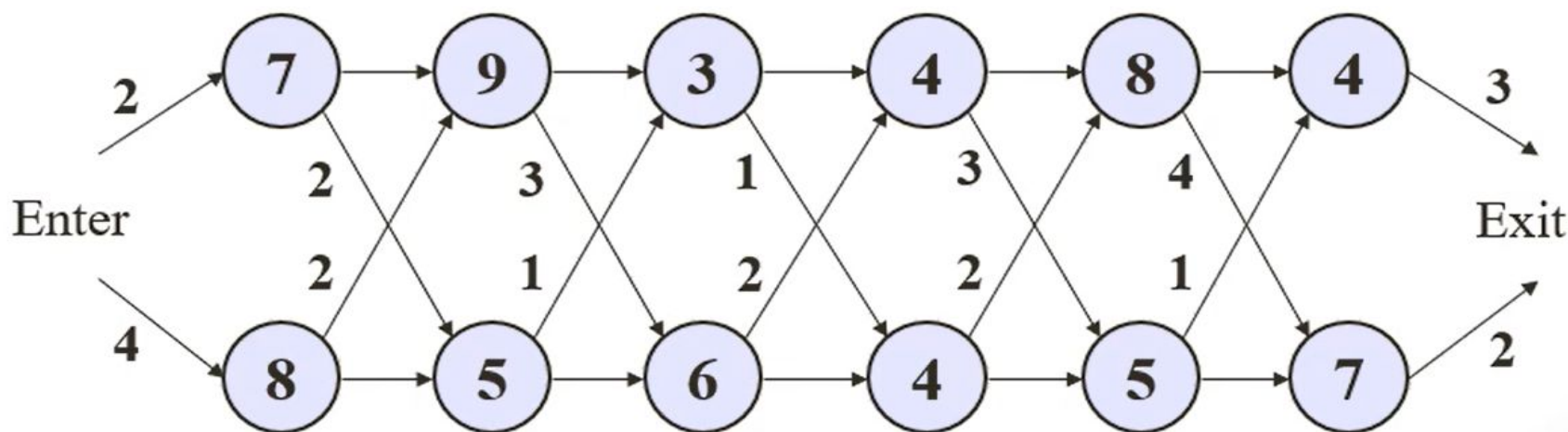
# Dynamic Programming: Assembly Line Scheduling

## 4. Construct an Optimal Solution

1. Print-Stations ( $l, n$ )
2.  $i \leftarrow l^*$
3. print "line"  $i$  ", station"  $n$
4. for  $j \leftarrow n$  downto 2
5.     do  $i \leftarrow l_i[j]$
6.     print "line"  $i$  ", station"  $j - 1$

	1	2	3	4	5	6
$l_1[j]$	1	1	2	1	1	2
$l_2[j]$	2	1	2	1	2	2

$l^* = 1$



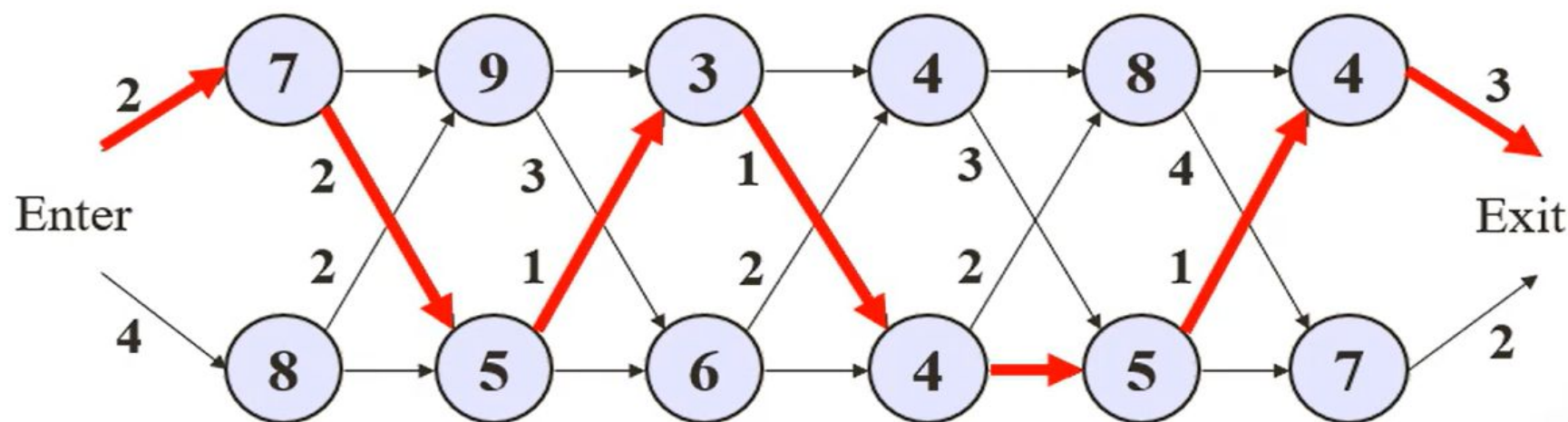
# Dynamic Programming: Assembly Line Scheduling

## 4. Construct an Optimal Solution

1. Print-Stations ( $l, n$ )
2.  $i \leftarrow l^*$
3. print "line"  $i$  ", station"  $n$
4. for  $j \leftarrow n$  downto 2
5.     do  $i \leftarrow l_i[j]$
6.     print "line"  $i$  ", station"  $j - 1$

	1	2	3	4	5	6
$l_1[j]$	1	1	2	1	1	2
$l_2[j]$	2	1	2	1	2	2

$l^* = 1$





```
//C++ Program of Assembly line Algorithm
#include <iostream>
#include <vector>
using namespace std;
int fun(vector<vector<int> > a, vector<vector<int> > t,
        int cl, int cs, int x1, int x2, int n)
{
    // base case
    if (cs == n - 1) {
        if (cl == 0) { // exiting from (current) line =0
            return x1;
        }
        else // exiting from line 2
            return x2;
    }
    // continue on same line
    int same
        = fun(a, t, cl, cs + 1, x1, x2, n) + a[cl][cs + 1];
    // continue on different line
    int diff = fun(a, t, !cl, cs + 1, x1, x2, n)
        + a[!cl][cs + 1] + t[cl][cs + 1];
    return min(same, diff);
}
```

```
int main()
{
    int n = 4; // number of statin
    vector<vector<int> > a
        = { { 4, 5, 3, 2 }, { 2, 10, 1, 4 } };
    vector<vector<int> > t
        = { { 0, 7, 4, 5 }, { 0, 9, 2, 8 } };
    int e1 = 10;
    int e2 = 12;
    int x1 = 18;
    int x2 = 7;
    // entry from 1st line
    int x = fun(a, t, 0, 0, x1, x2, n) + e1 + a[0][0];
    // entry from 2nd line
    int y = fun(a, t, 1, 0, x1, x2, n) + e2 + a[1][0];
    cout << min(x, y) << endl;
}
```

## Brute Force Algorithm:

This is the most basic and simplest type of algorithm. A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

A brute force algorithm is a simple, comprehensive search strategy that systematically explores every option until a problem's answer is discovered. It's a generic approach to problem-solving that's employed when the issue is small enough to make an in-depth investigation possible. However, because of their high temporal complexity, brute force techniques are inefficient for large-scale issues.

### Example:

If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.

## Key takeaways:

**Methodical Listing:** Brute force algorithms investigate every potential solution to an issue, usually in an organized and detailed way. This involves attempting each option in a specified order.

**Relevance:** When the issue space is small and easily explorable in a fair length of time, brute force is the most appropriate method. The temporal complexity of the algorithm becomes unfeasible for larger issue situations.

**Not using optimization or heuristics:** Brute force algorithms don't use optimization or heuristic approaches. They depend on testing every potential outcome without ruling out any using clever pruning or heuristics.

# What is Travelling Salesman Problem?

The Travelling Salesperson Problem (TSP) is an optimization problem where a salesperson must visit a given set of cities exactly once, starting and ending at the same city. The goal is to find the shortest possible route that covers all the cities and returns to the starting point.

## Why is TSP Algorithm Important?

The algorithm for salesman Travelling problem is important because it represents real-world problems that many industries face, such as:

Delivery services need to plan routes to drop off packages at multiple locations.

Logistics companies need to find the shortest path to transport goods efficiently.

Manufacturing companies need to reduce the travel of robotic arms in factories.

Solving TSP helps reduce time, fuel costs, and energy, making operations faster and cheaper.

# Travelling Salesman Problem: Example

Let's say we have a salesperson who needs to visit four popular Indian cities: Mumbai, Delhi, Bengaluru, and Chennai, and they want to find the shortest route that visits each city exactly once and returns to the starting city.

The distances between the cities are:

	Mumbai	Delhi	Bengaluru	Chennai
Mumbai	0	1,400 km	980 km	1,330 km
Delhi	1,400 km	0	2,150 km	2,200 km
Bengaluru	980 km	2,150 km	0	350 km
Chennai	1,330 km	2,200 km	350 km	0

**Problem:** The salesperson starts in Mumbai and must visit Delhi, Bengaluru, and Chennai exactly once, then return to Mumbai. The goal is to find the shortest route.

**Possible Routes:** Let's calculate the total distance for a few possible routes:

**1. Mumbai → Delhi → Bengaluru → Chennai → Mumbai**

- Mumbai → Delhi = 1,400 km
- Delhi → Bengaluru = 2,150 km
- Bengaluru → Chennai = 350 km
- Chennai → Mumbai = 1,330 km

**Total distance** =  $1,400 + 2,150 + 350 + 1,330 = 5,230$  km

**2. Mumbai → Delhi → Chennai → Bengaluru → Mumbai**

Mumbai → Delhi = 1,400 km

Delhi → Chennai = 2,200 km

Chennai → Bengaluru = 350 km

Bengaluru → Mumbai = 980 km

**Total distance** =  $1,400 + 2,200 + 350 + 980 = 4,930$  km

### 3. Mumbai → Bengaluru → Chennai → Delhi → Mumbai

Mumbai → Bengaluru = 980 km

Bengaluru → Chennai = 350 km

Chennai → Delhi = 2,200 km

Delhi → Mumbai = 1,400 km

**Total distance** =  $980 + 350 + 2,200 + 1,400 = 4,930$  km

**The shortest routes are:**

Mumbai → Delhi → Chennai → Bengaluru → Mumbai with a total distance of 4,930 km.

Mumbai → Bengaluru → Chennai → Delhi → Mumbai with a total distance of 4,930 km.

In both cases, the total travel distance is the same, and this is the most efficient route for the salesperson to minimize travel distance.



# Job Assignment Problem

- Let us consider that there are  $n$  people and  $n$  jobs.
- Each person has to be assigned only one job.
- When the  $j^{\text{th}}$  job is assigned to  $p^{\text{th}}$  person, the cost incurred is represented by  $C$ .

$$C = C[p, j]$$

where,  $p = 1, 2, 3, \dots, n$  and  $j = 1, 2, 3, \dots, n$

- The number of permutations (the number of different assignments to different persons) is  $n!$
- The brute force approach is impractical for large value of  $n$ .

# Example Problem

	Job1	Job2	Job3
Person1	4	7	2
Person2	3	5	8
Person3	9	1	6

# Finding all the possible solutions

Group1

P1	P2	P3
J1	J2	J3

$$\text{Cost}=4+5+6=15$$

P1	P2	P3
J1	J3	J2

$$\text{Cost}=4+8+1=13$$

Group 2

P1	P2	P3
J2	J1	J3

$$\text{Cost}=7+3+6=16$$

P1	P2	P3
J2	J3	J1

$$\text{Cost}=7+8+9=24$$

Group 3

P1	P2	P3
J3	J1	J2

$$\text{Cost}=2+3+1=6$$

P1	P2	P3
J3	J2	J1

$$\text{Cost}=2+5+9=16$$

# Optimum solution in each group

- Group 1

P1	P2	P3
J1	J3	J2

$$\text{Cost} = 4 + 8 + 1 = 13$$

- Group 2

P1	P2	P3
J2	J1	J3

$$\text{Cost} = 7 + 3 + 6 = 16$$

- Group 3

P1	P2	P3
J3	J1	J2

$$\text{Cost} = 2 + 3 + 1 = 6$$

# Optimum Solution and cost

P1	P2	P3
J3	J1	J2

**Optimum  
Cost=2+3+1=6**

**Efficiency  $O(n!)$**

The background features abstract, overlapping geometric shapes in various shades of green, ranging from light lime to dark forest green. These shapes are primarily located on the left and right sides of the frame, creating a modern, layered effect. The central area is a plain white background.

**END**