Dynamic Programming: Introduction

- An **optimization problem** is one that has multiple feasible solutions, each having a specific cost. Our objective is to find the best of all possible solutions.

- **Dynamic Programming** is typically used to solve optimization problems.

- Introduced by Richard Bellman in 1955, the word **dynamic** reflects the time-varying aspect of the problems. The word **programming** refers to tabular method (like linear programming) to solve a problem. Doesn't really refer to computer programming.

- Solves every sub problem just once and save the solution in a table. Avoids recomputing the solution every time the sub-problem is encountered.

- Its not actually an algorithm but a meta-technique/strategy for designing algorithms.

Dynamic Programming: Properties

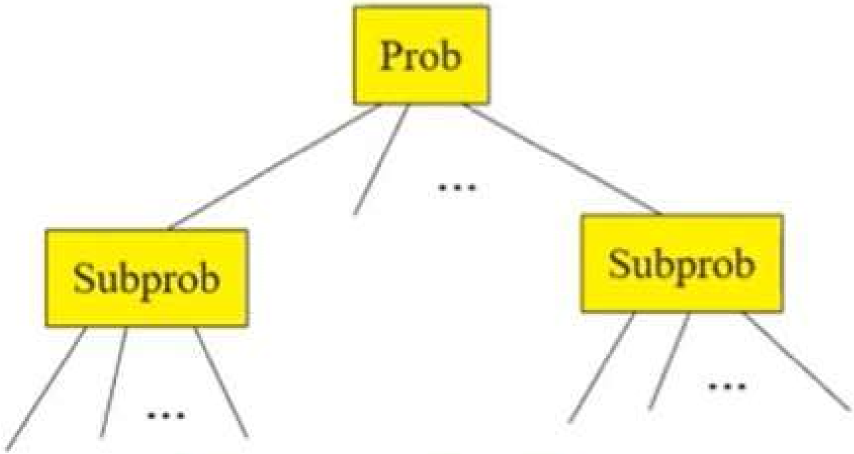
- How to know if an optimization problem can be solved by applying dynamic programming?

- Dynamic programming works if a problem exhibits the following two properties:
  - **Optimal sub-structure:** An optimal solution to the problem contains within it, optimal solutions to sub-problems
  - **Overlapping sub-problems:** When a recursive algorithm revisits the same problem repeatedly, then the optimization problem has overlapping sub-problems

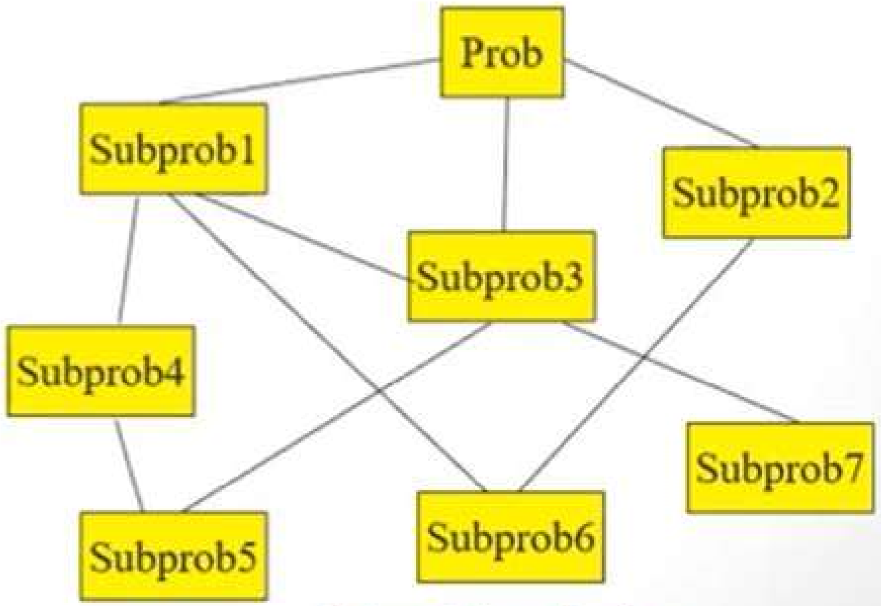
Dynamic Programming: Properties

- Dynamic programming, like divide-and-conquer method, solves problems by combining the solutions to sub-problems.

- Divide-and-conquer algorithms have ***independent*** sub-problems while dynamic programming is applicable when the sub-problems are ***not independent***.



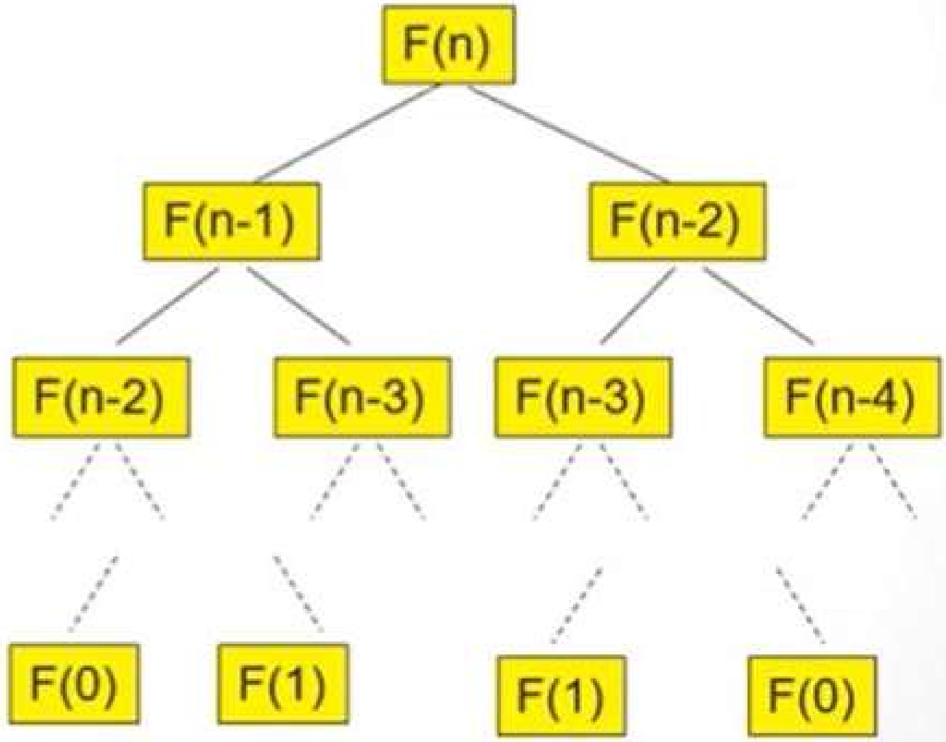
Independent sub-problems
(no overlapping work).



Subproblem sharing

- What is the drawback of recursive algorithms like Fibonacci sequence?

```
Fib(n)
    if ( n <= 1)
        return n;
    return ( Fib(n-1) + Fib(n-2));
```

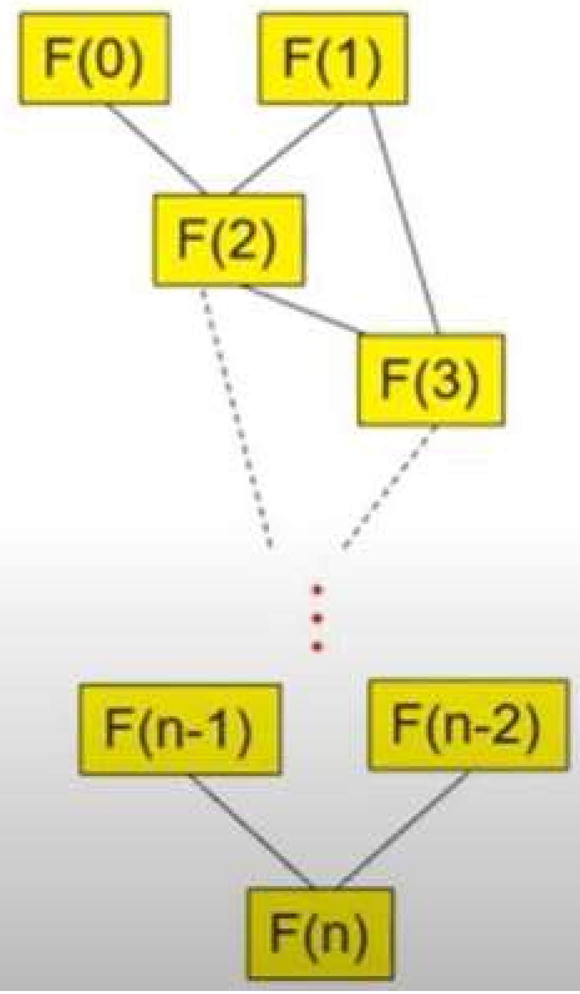


$O(2^n)$

- Solving Fibonacci sequence using dynamic programming.

```
Fib_DP(n)
    f[0] = 0;
    f[1] = 1;
    for i = 2 to n
        f[i] = f[i-1] + f[i-2];
    return f[n];
```

$O(n)$



Dynamic Programming: Steps

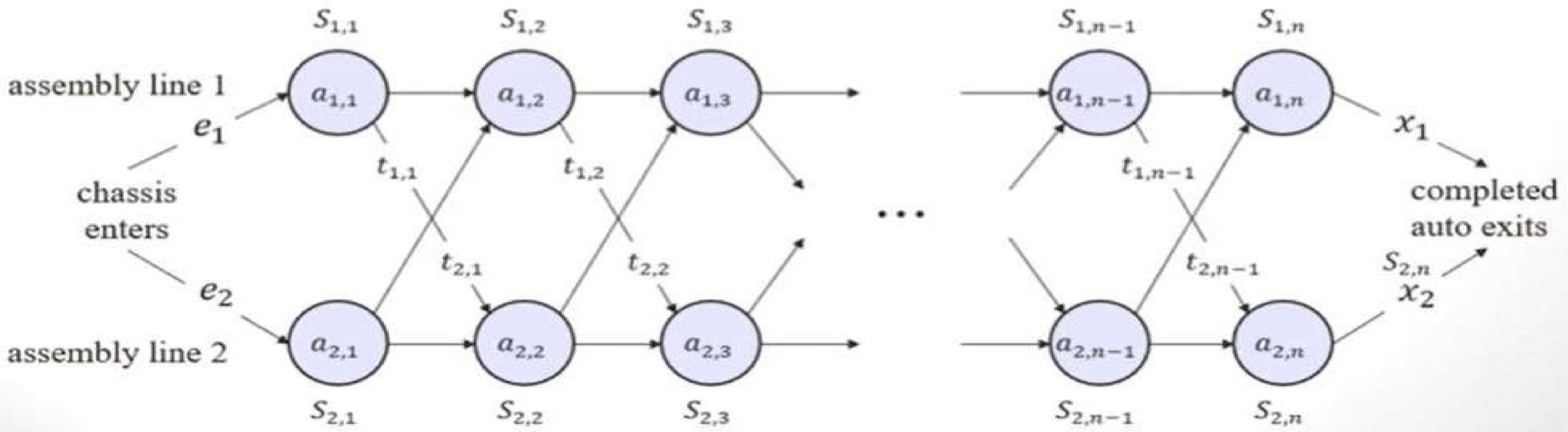
- Steps to design a Dynamic Programming algorithm:
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution, typically in a bottom-up fashion
  4. Construct an optimal solution from computed information

Problems

- Assembly Line Scheduling
- Matrix Chain Multiplication
- Longest Common Subsequence
- 0-1 Knapsack
- TSP

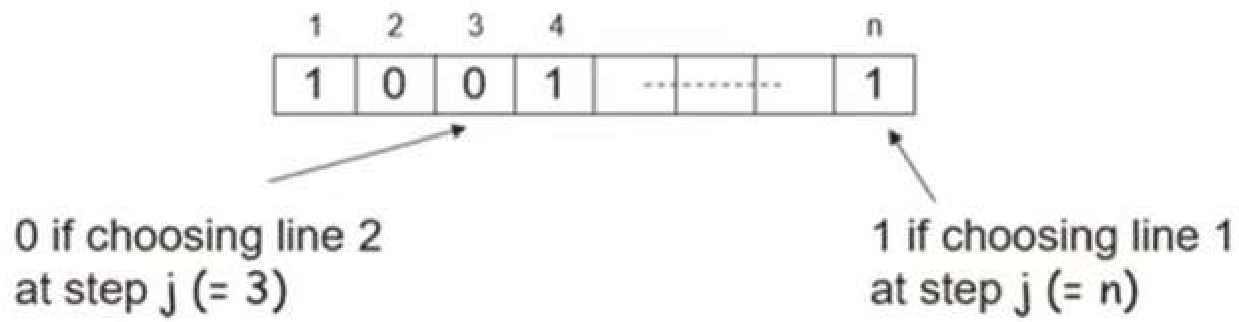
Dynamic Programming: Assembly Line Scheduling

- Automobile factory has two assembly lines
  - Each line has n stations: $S_{1,1}, \cdots, S_{1,n}$ and $S_{2,1}, \cdots, S_{2,n}$.
  - 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

- Brute force solution:
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one



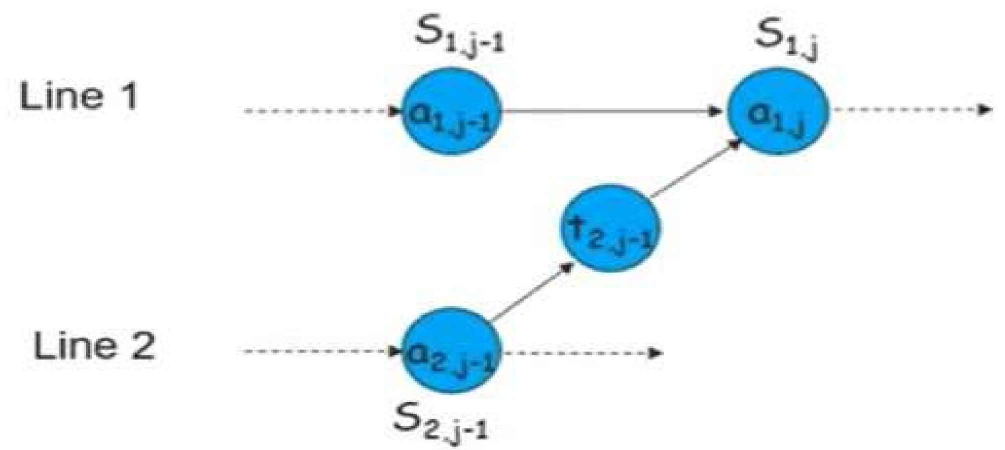
- There are 2^n possible ways to choose stations
- Infeasible when n is large!!

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

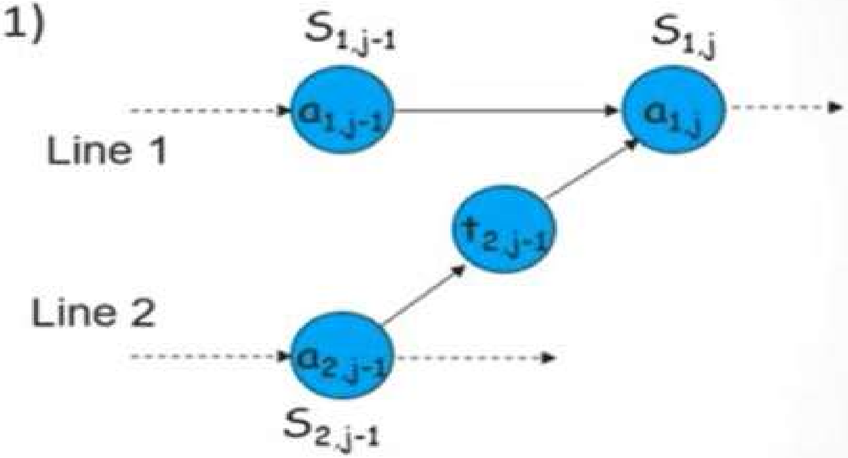
- We define the value of an optimal solution in terms of the optimal solution to subproblems

- **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 $S_{i,j}$.

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

Dynamic Programming: Assembly Line Scheduling

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:** $j = 2, 3, \cdots, n, and\ i = 1, 2$
- 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

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

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

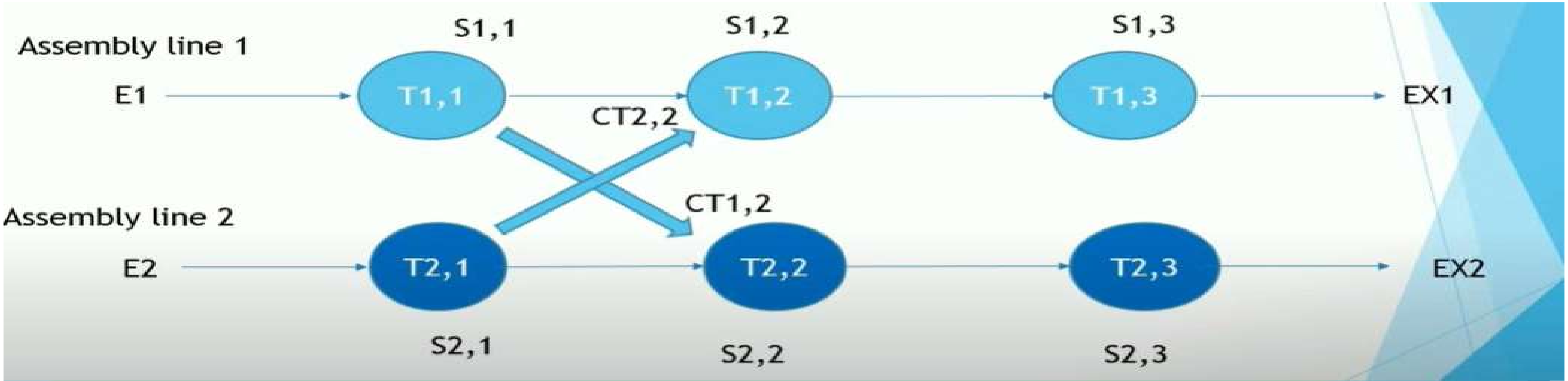
Dynamic Programming: Assembly Line Scheduling

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:

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



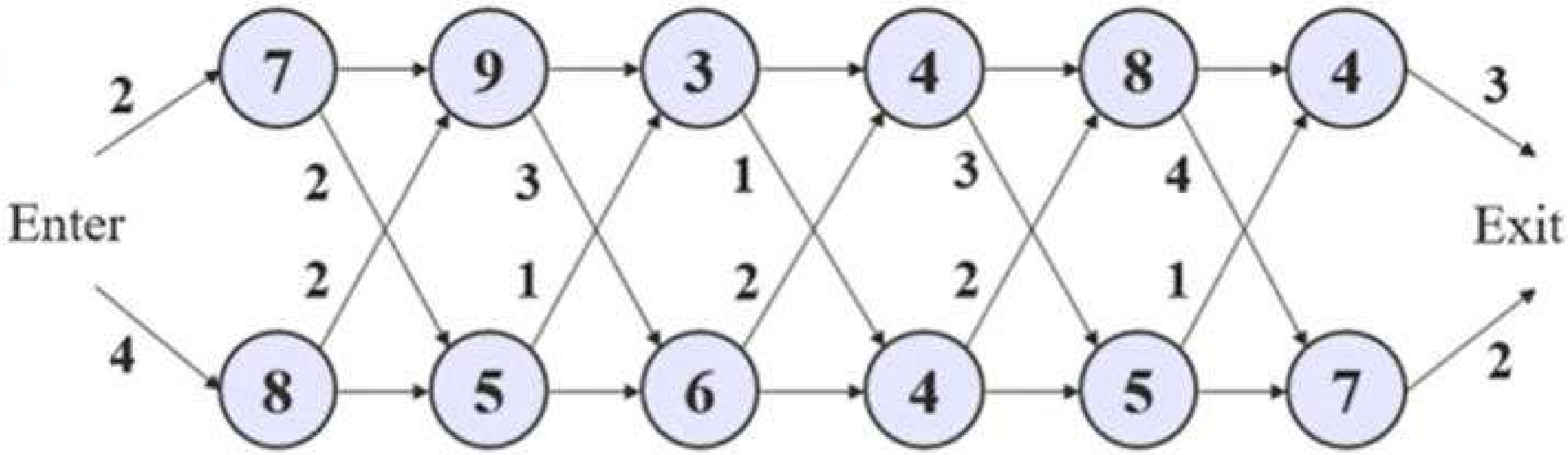
T1[0] = e[0]+t[0][0]
T2[0] =e[1]+t[1][0]

T1[i]=min(T1[i−1]
+ a[0][i],T2[i−1]+ct[1][i] + a[0][i])

T2[i] = min(T2[i−1] + a[1][i],
T1[i−1] + ct[0][i]+ a[1][i])

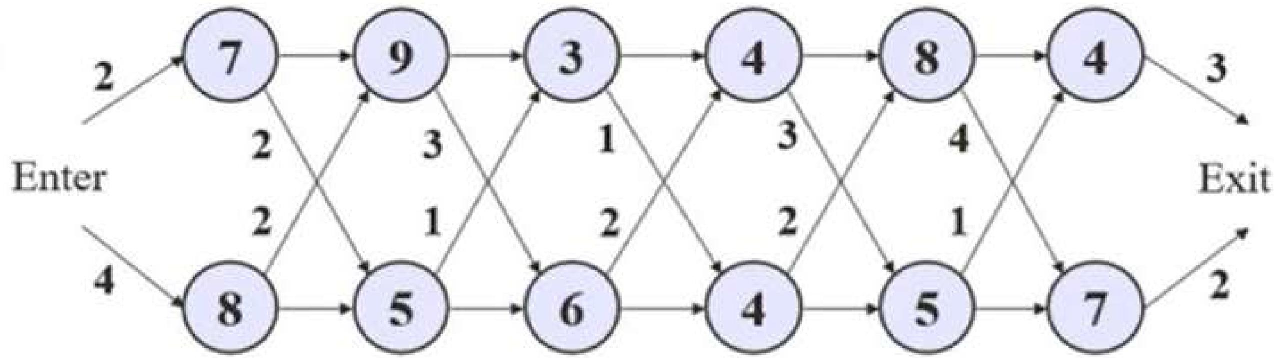
min_time = min(T1[n−1]+x1[0],T2[n−1]+x2[0])

Example:



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
$f_1[j]$	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

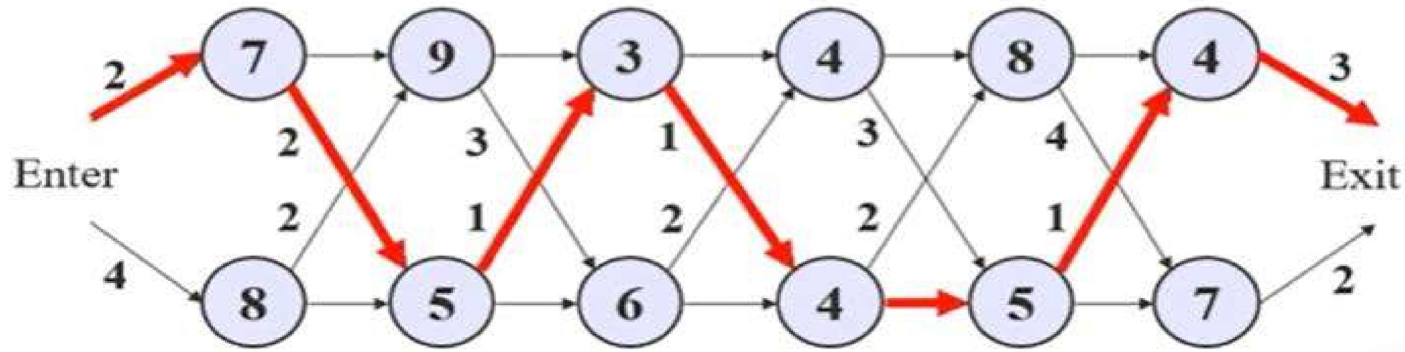
Dynamic Programming: Assembly Line Scheduling

4. Construct an Optimal Solution

```
Print-Stations (l, n)
    i ← l*
    print "line" i ", station" n
    for j ← n downto 2
        do i ← l_i[j]
            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

$l^* = 1$



ALGORITHM

```
FASTEST-WAY(a, t, e, x, n)
    f1[1] ← e1 + a1,1
    f2[1] ← e2 + a2,1
```

Compute initial values of f_1 and f_2

```
    for j ← 2 to n
        do if f1[j - 1] + a1,j ≤ f2[j - 1] + t2,j-1 + a1,j
            then f1[j] ← f1[j - 1] + a1,j
                 l1[j] ← 1
            else f1[j] ← f2[j - 1] + t2,j-1 + a1,j
                 l1[j] ← 2
```

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

```
           if f2[j - 1] + a2,j ≤ f1[j - 1] + t1,j-1 + a2,j
            then f2[j] ← f2[j - 1] + a2,j
                 l2[j] ← 2
            else f2[ j] ← f1[ j - 1] + t1,j-1 + a2,j
                 l2[ j] ← 1
```

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

```
    if f1[n] + x1 ≤ f2[n] + x2
     then f* = f1[n] + x1
          l* = 1
     else f* = f2[n] + x2
          l* = 2
```

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

complexity

The time and Space complexity of the algorithm:

$$\mathbf{O(n)}$$