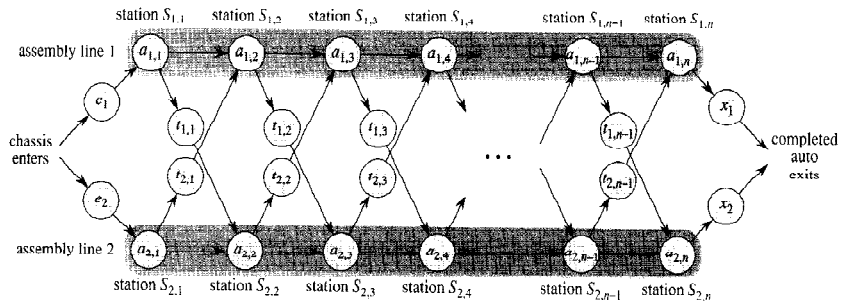
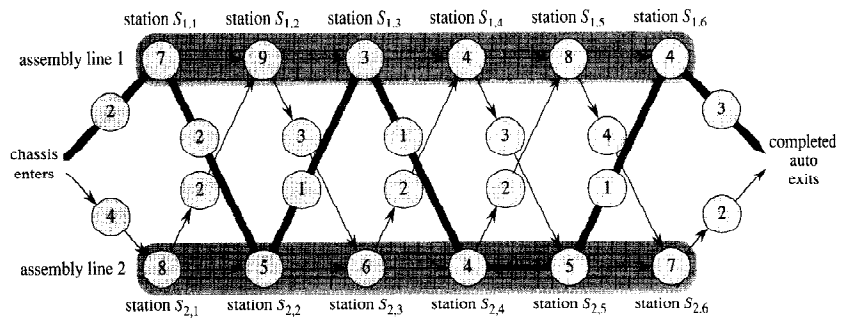


# Assembly-Line Scheduling

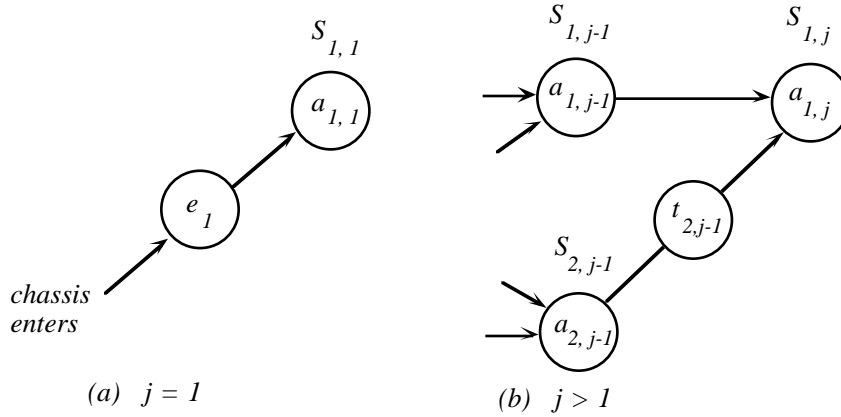


# Instance of Assembly-Line Scheduling



(a)

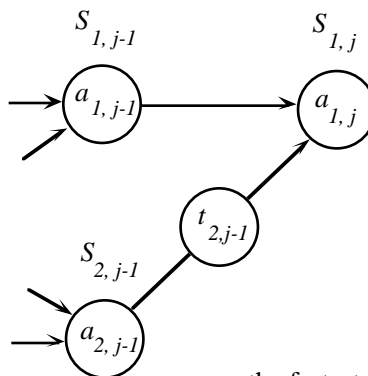
## Step 1. Structure of Scheduling Problem\*



\*We consider the fastest possible way for a chassis to get from the starting point through station  $S_{1,j}$

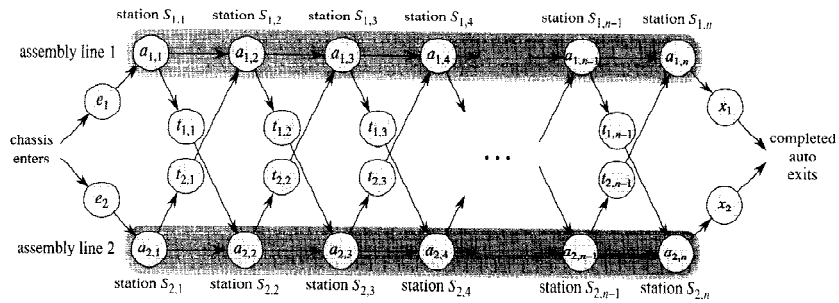
## The fastest way through $S_{1,j}$ is either ...

... the fastest way through station  $S_{1,j-1}$  and then directly through station  $S_{1,j}$  or



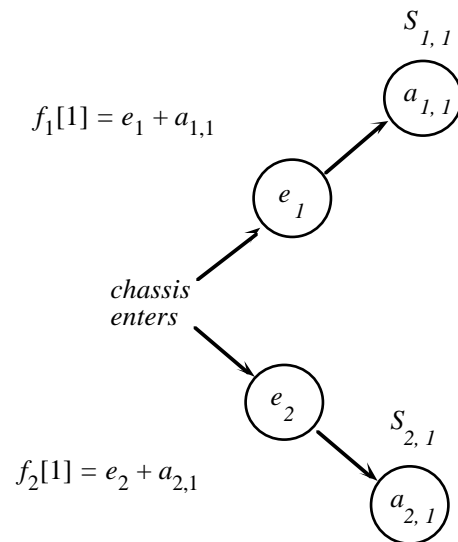
... the fastest way through station  $S_{2,j-1}$ , a transfer to line 1, and then through station  $S_{1,j}$ .

## Step 2. A Recursive Solution\*

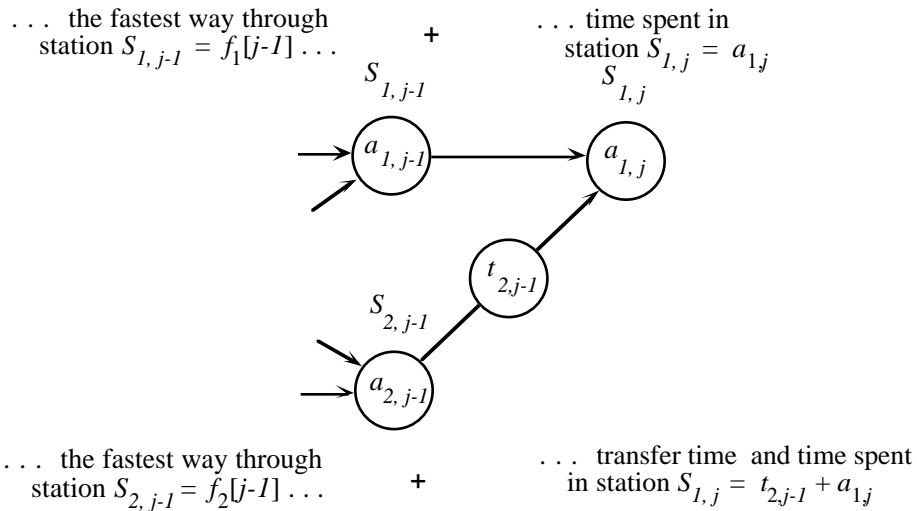


\* We consider the fastest possible way for a chassis to get from the starting point through station  $S_{i,j}$  for  $i = 1$  or  $2$  and  $j = 1, 2, \dots, n$ .

## To get through station 1 on either line ...



## To find $f_1[j]$ for $j > 1$ , take smaller of . . .



\*We say that the problem exhibits the *optimal substructure* property.

## Fastest times through station $j$ on either line

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

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

\*The fastest way through the entire factory  $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$ .

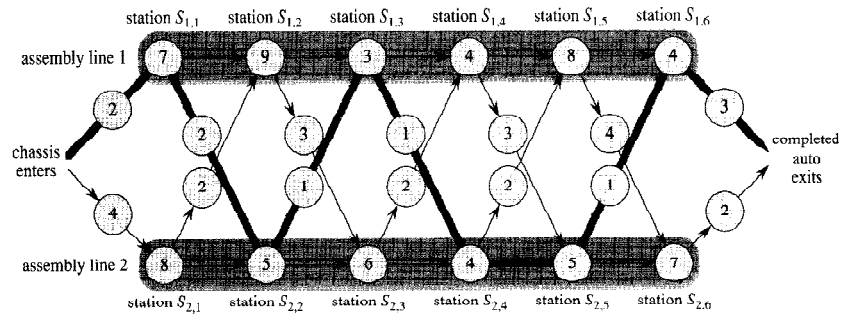
## A back of the envelope calculation . . .

Let  $r_i(j)$  be the number of times  $f_i[j]$  is calculated using the recursive equations on the previous slide. Then,

$$r_i(j) = \begin{cases} 1 & \text{if } j = n \\ r_1(j+1) + r_2(j+1) & \text{if } 1 \leq j < n \end{cases}$$

\*We show  $r_i(j) = 2^{n-j}$ , for  $i = 1, 2$ , by induction on  $n - j$ .

## There is another way,



(a)

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

$f^* = 38$

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

$l^* = 1$

(b)

\*Where  $l_i[j]$  is the line number, 1 or 2, whose station  $j-1$  is used in fastest way through station  $S_{i,j}$

## Dynamic Programming Approach\*

---

```
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  $f^* \leftarrow f_1[n] + x_1$ 
16          $l^* \leftarrow 1$ 
17     else  $f^* \leftarrow f_2[n] + x_2$ 
18          $l^* \leftarrow 2$ 
```

\*Complexity?

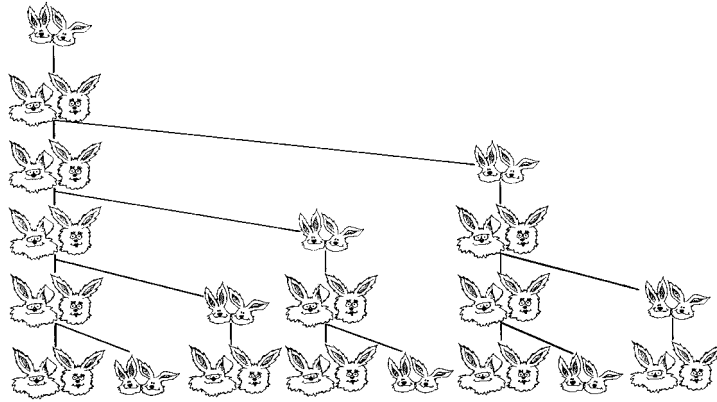
## Think dynamic programming if problem ...

---

- ... exhibits *optimal substructure* property,  
i.e., optimal solution contains optimal solutions  
to subproblems, and
- ... has *overlapping subproblems*,  
i.e., a standard recursive solution revisits the same  
subproblem over and over again.

## A Simpler Example

---

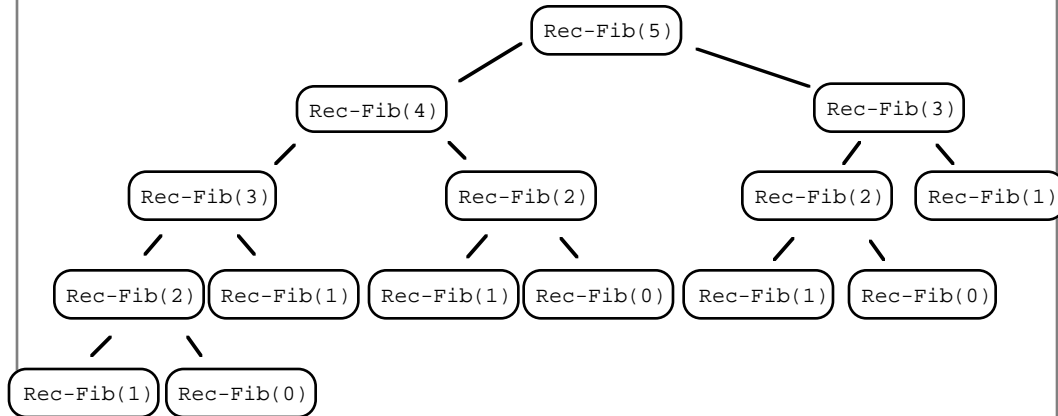


## Recursive Fibonacci

---

```
Rec-Fib(n)  
  if n ≤ 1  
    then return n  
    else return Rec-Fib(n-1)+Rec-Fib(n-2)
```

## Invocation Tree for Rec-Fib



\*Estimated running time of Rec-Fib?

## Dynamic Programming Approach

```
Dynamic-Fib(n)
  Fib ← new array [0..n]
  Fib[0] ← 0
  Fib[1] ← 1
  for i ← 2 to n
    do Fib[i] ← Fib[i-1] + Fib[i-2]
  return Fib[n]
```

n	result
0	0
1	1
2	
3	
4	
5	

Dynamic Programming Array FIB

\*Estimated running time of Dynamic-Fib?

## The Best of Both Worlds

---

*Memoization* offers the efficiency of dynamic programming while maintaining a top-down strategy.

```
Rec-Fib(n)
  if n ≤ 1
    then return n
  else return Rec-Fib(n-1)+Rec-Fib(n-2)
```

A *memoized* recursive algorithm maintains an entry in a table for the solution to each subproblem. Entries are filled out when the corresponding subproblem is first encountered.

## Memoized Recursive Fibonacci

---

```
Fast-Fib(n)
  T ← new array[0..n]
  for i ← 2 to n
    do T[i] ← -1
  return Mem-Fib(T, n)
```

```
Mem-Fib(T, n)
  if T[n] = -1
    then
      if n ≤ 1
        then T[n] ← n
      else T[n] ← Mem-Fib(T, n-1) + Mem-Fib(T, n-2)
  return T[n]
```

n	result
0	
1	
2	
3	
4	
5	

Memoization array *T*