INT102
Algorithmic Foundations and Problem Solving

Dynamic Programming

Dr Pengfei Fan
Department of Intelligent Science

西交利物浦大学
Xi'an Jiaotong-Liverpool University

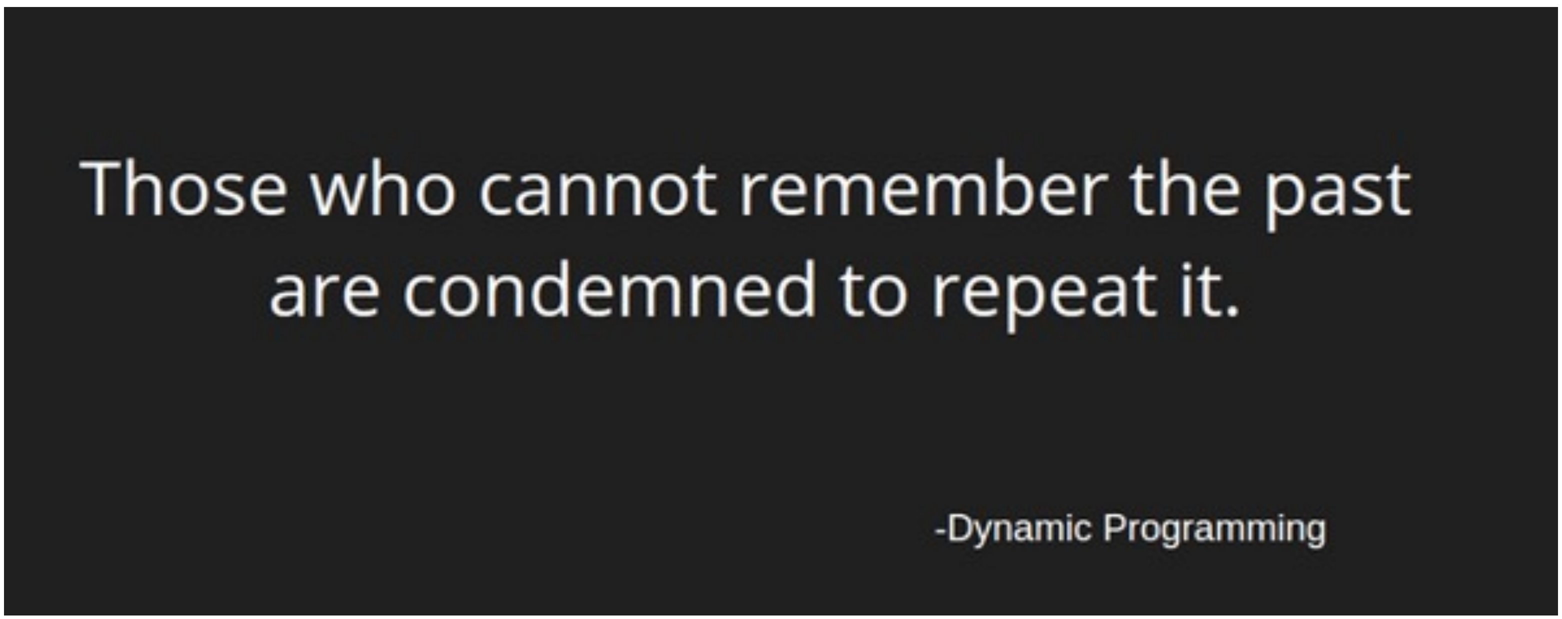
Dynamic programming

Learning outcomes

- Understand the basic idea of dynamic programming
- Able to apply dynamic programming to compute Fibonacci numbers
- Able to apply dynamic programming to solve the assembly line scheduling problem

Dynamic programming

an efficient way to implement some divide and conquer algorithms



Dynamic programming

- The basic steps of dynamic programming are as follows:
  - Define the problem and identify the sub-problems.
  - Formulate a recursive solution to the problem.
  - Memoize the solutions of the sub-problems in a table or array.
  - Use the memoized solutions to compute the optimal solution to the problem.

Fibonacci numbers

Problem with recursive method

Fibonacci number F(n)

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10
F(n)	1	1	2	3	5	8	13	21	34	55	89

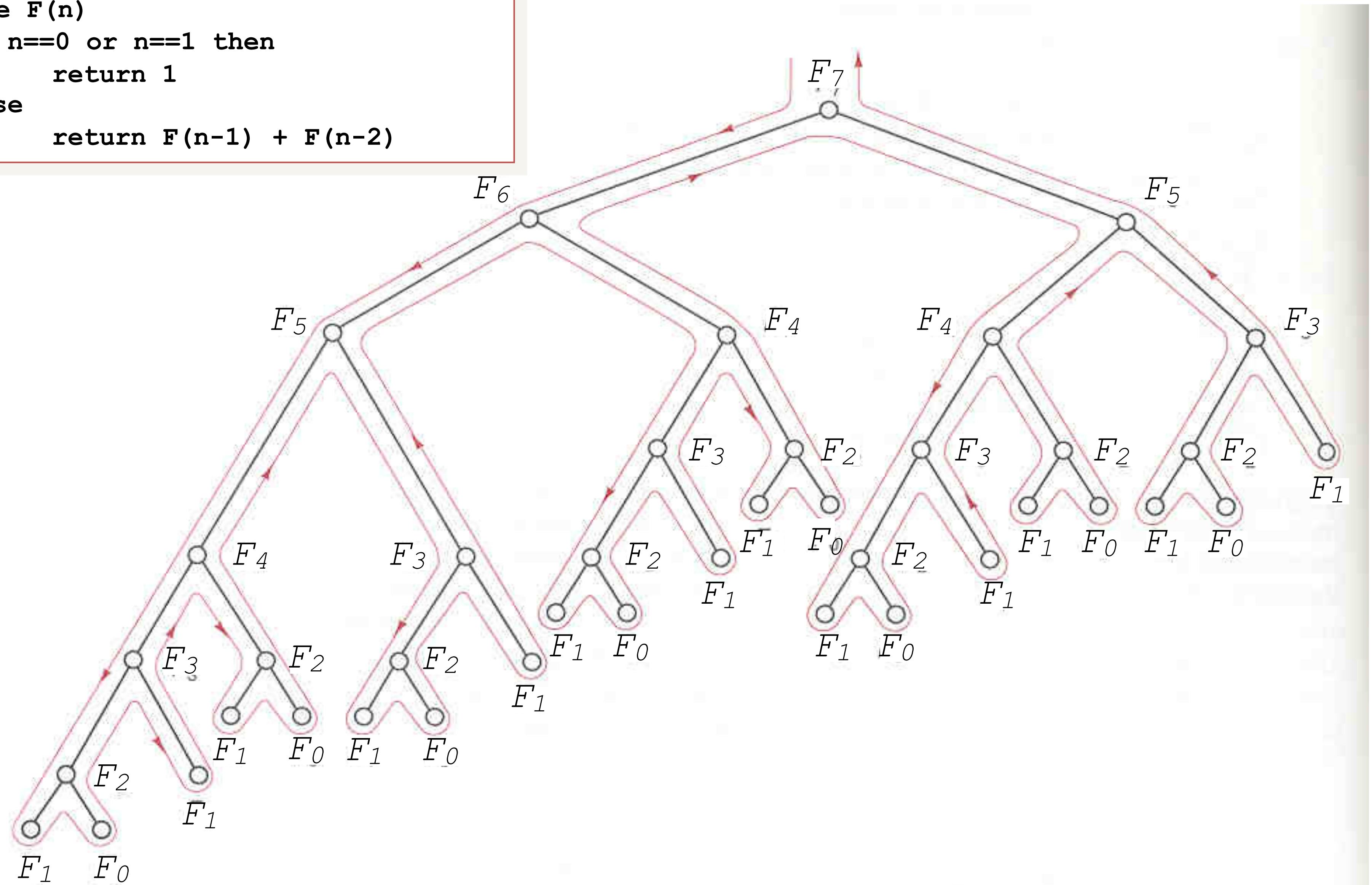
Two approaches

```
Procedure F(n)
   if n==0 or n==1 then
      return 1
   else
      return F(n-1) + F(n-2)
```

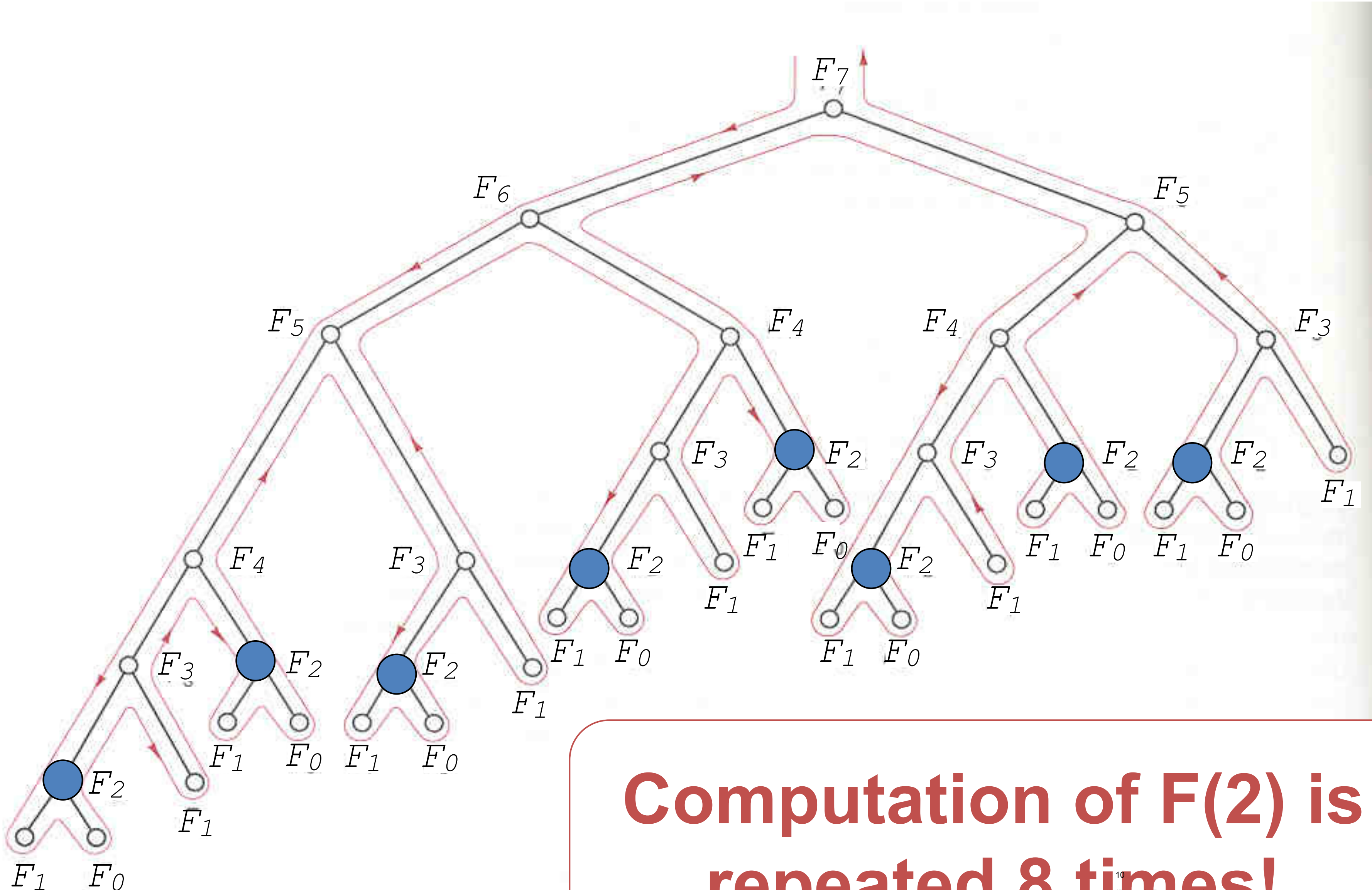
```
Procedure F(n)
   Set A[0] = A[1] = 1
   for i = 2 to n do
      A[i] = A[i-1] + A[i-2]
   return A[n]
```

The execution of F(7)

```
Procedure F(n)
      if n==0 or n==1 then
            return 1
      else
            return F(n-1) + F(n-2)
```

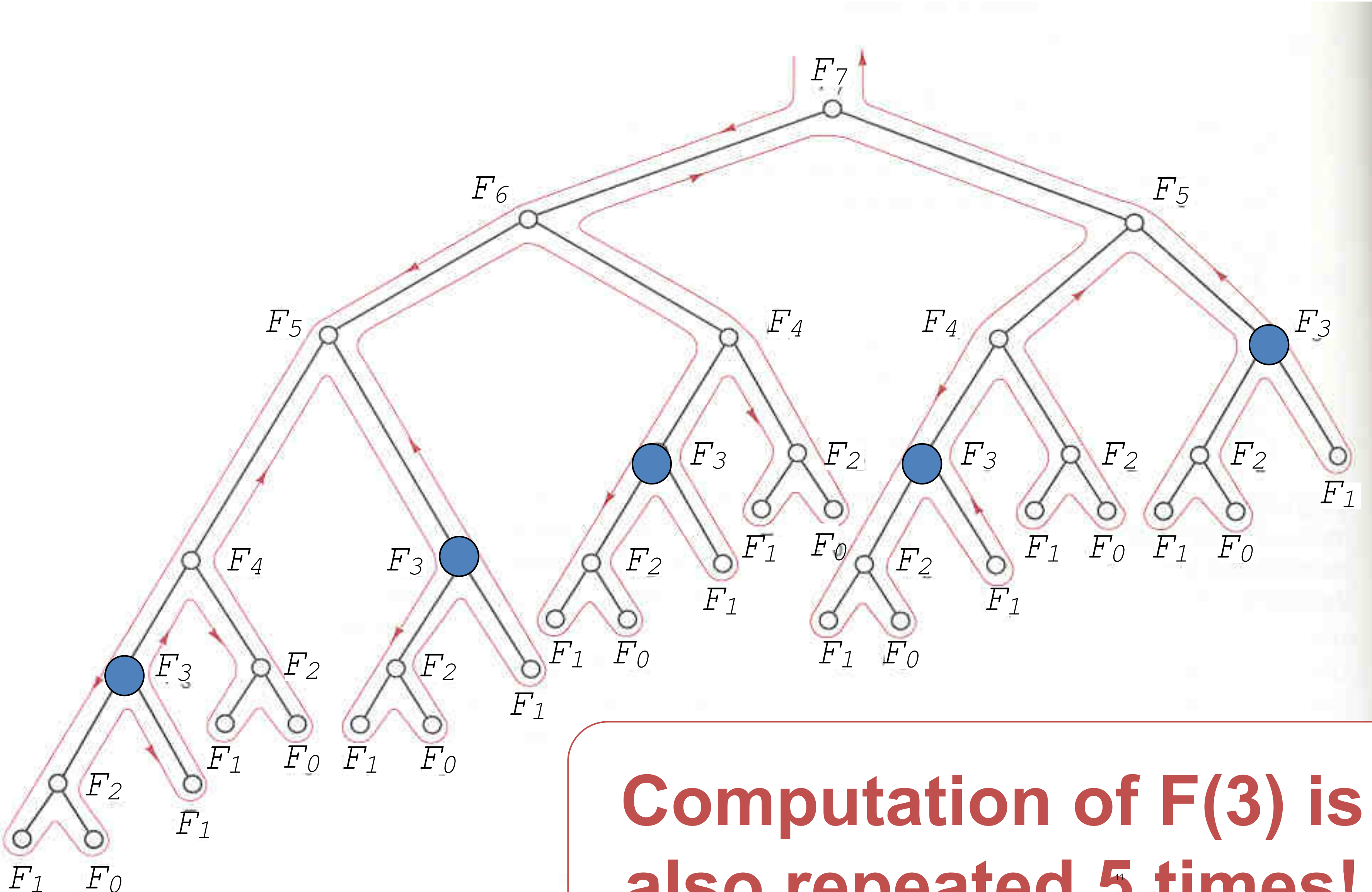


The execution of F(7)



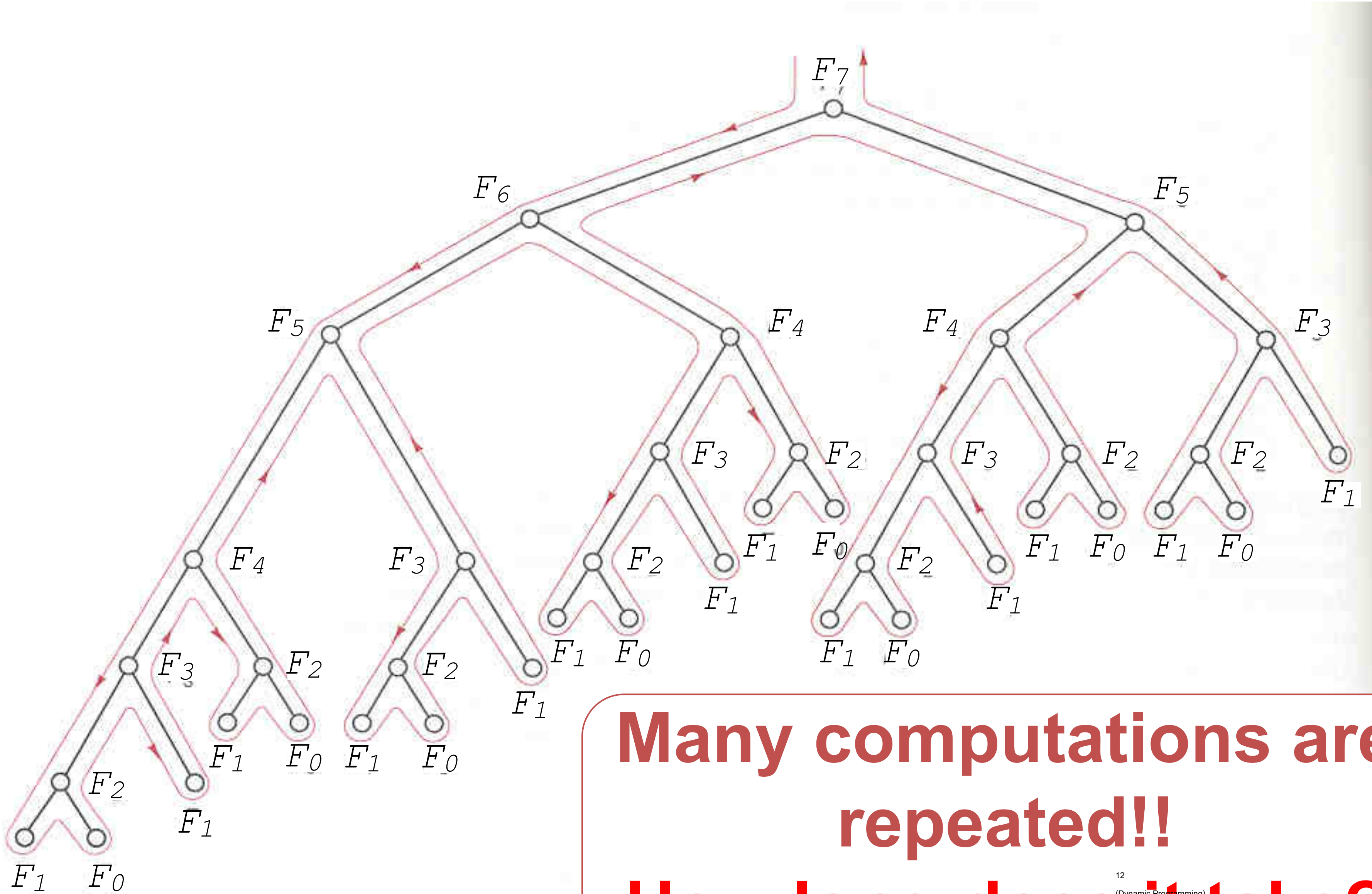
Computation of F(2) is repeated 8 times!

The execution of F(7)



Computation of F(3) is also repeated 5 times!

The execution of F(7)



Many computations are repeated!!

How long does it take?

Recursive version - exponential

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2) + 1 \\
&= [f(n-2)+f(n-3)+1]+f(n-2)+1 \\
&> 2\, f(n-2) \\
&> 2\, [2\, f(n-2-2)] = 2^2\, f(n-4) \\
&> 2^2\, [2\, f(n-4-2)] = 2^3\, f(n-6) \\
&\ldots \\
&> 2^k\, f(n-2k)
\end{aligned}
$$

Suppose f(n) denote the time complexity to compute F(n)

exponential in n

If n is even, $f(n) > 2^{n/2} f(0) = 2^{n/2}$

If n is odd, $f(n) > f(n-1) > 2^{(n-1)/2}$

Can we avoid exponential time?

Idea for improvement

Memoization:

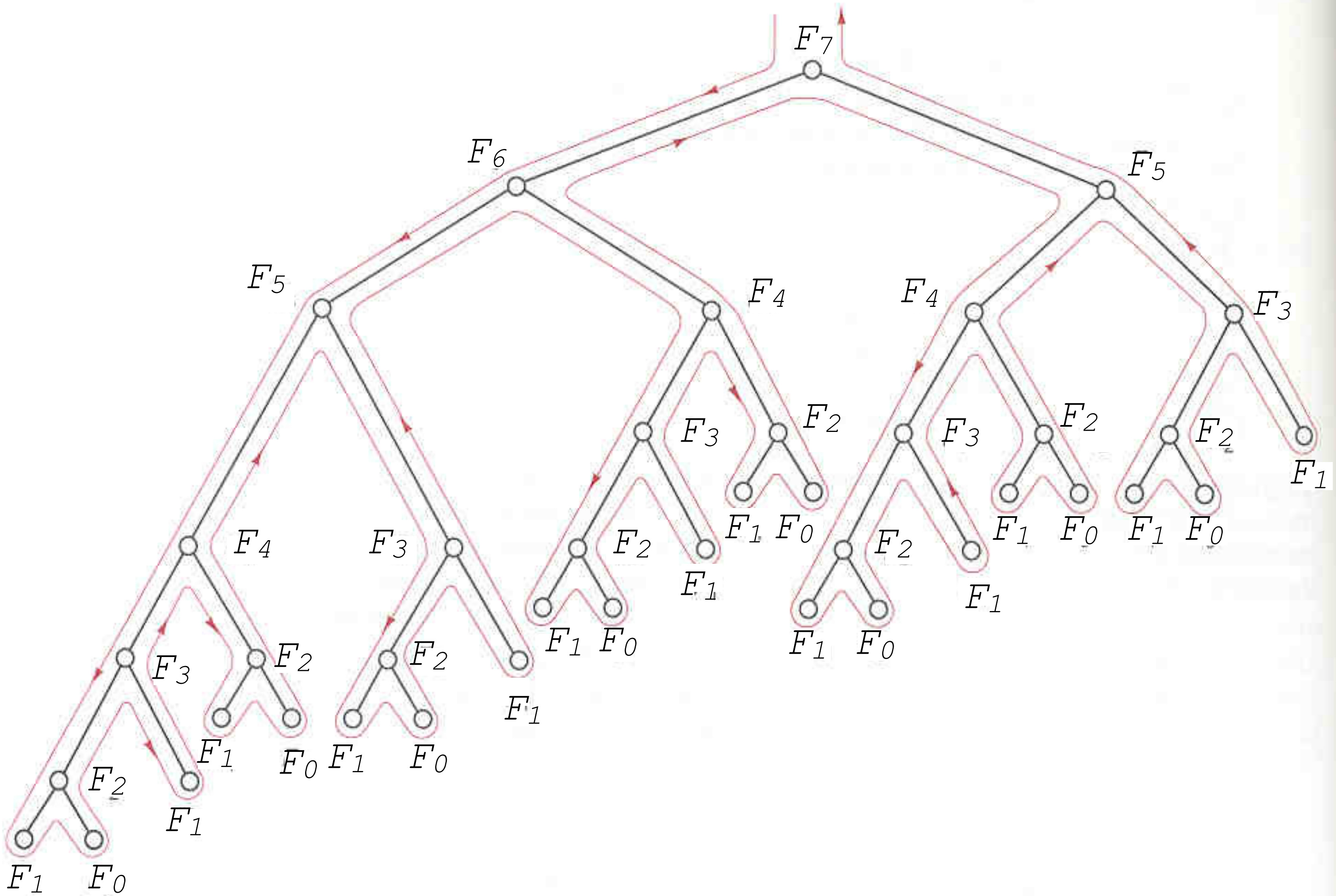
- Store F(i) somewhere after we have computed its value
- Afterward, we don't need to re-compute F(i); we can retrieve its value from our memory.

```
Procedure F(n)
  if (v[n] < 0) then
    v[n] =  F(n-1)+F(n-2)
  return v[n]
```

```
Main
  set v[0] = v[1] = 1
  for i = 2 to n do
    v[i] = -1
  output F(n)
```

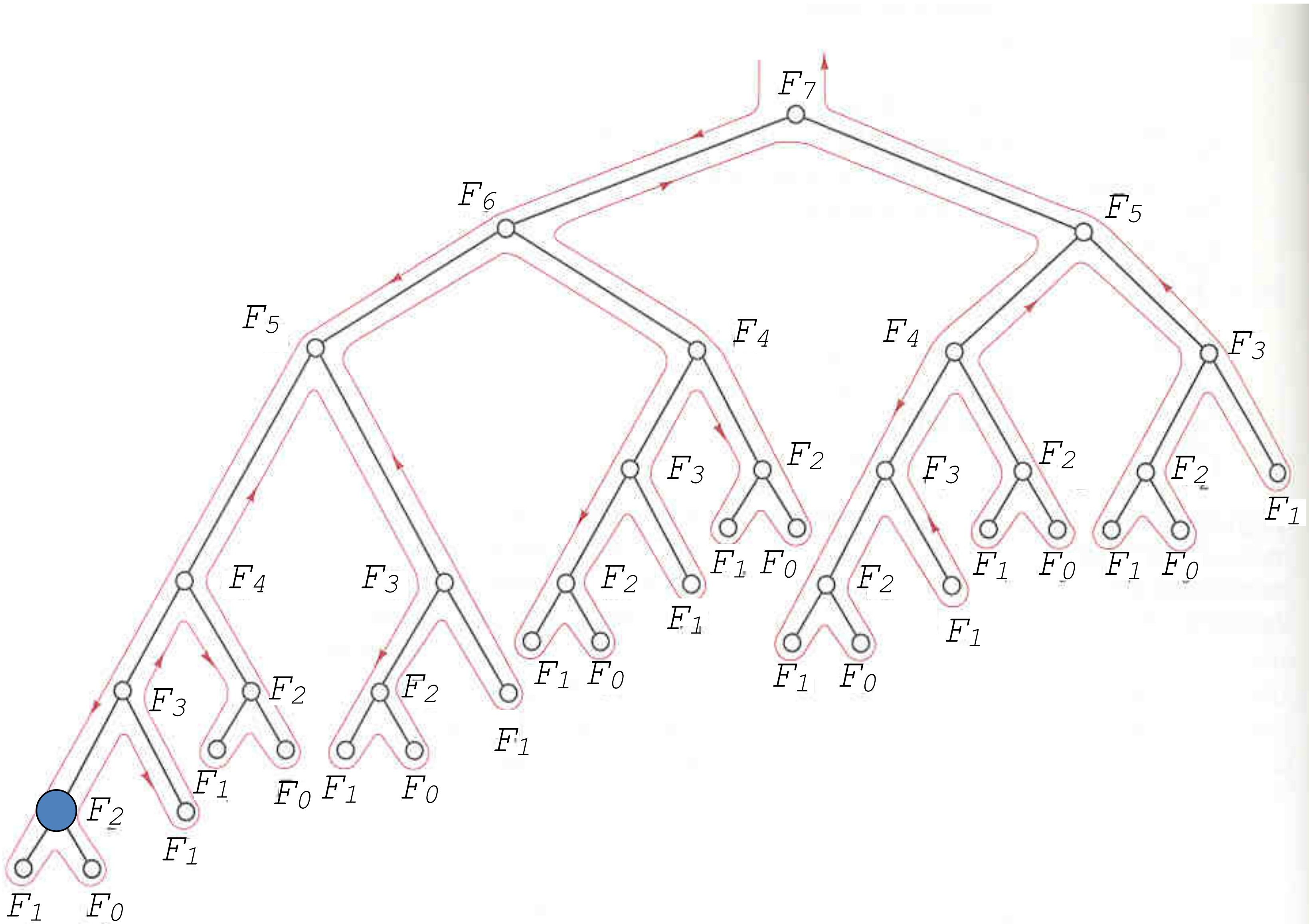
[ ] refers to array
( ) is parameter for calling a procedure

Look at the execution of F(7)



v[0]	1
v[1]	1
v[2]	-1
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)

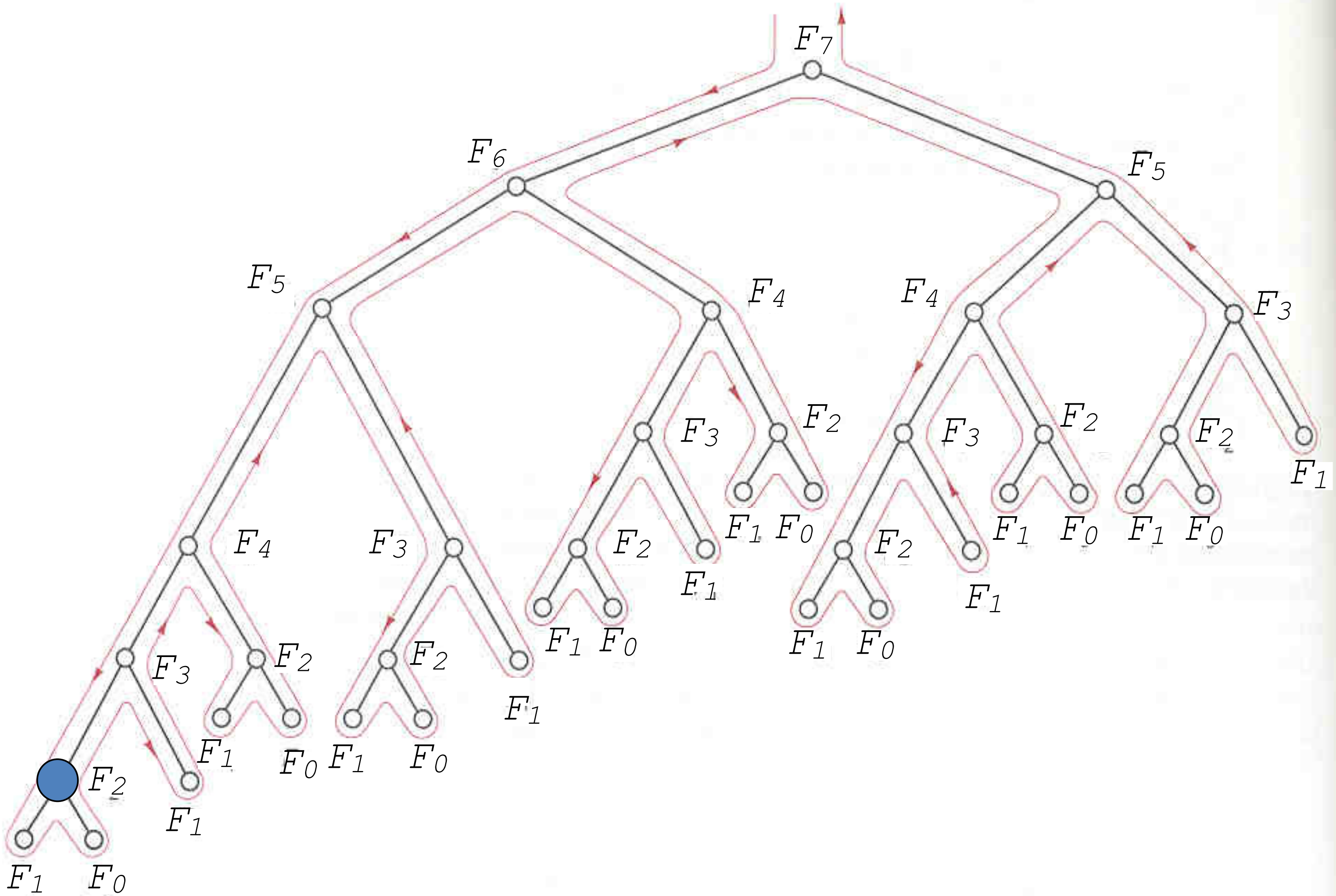


16

(Dynamic Programming)

v[0]	1
v[1]	1
v[2]	-1
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)

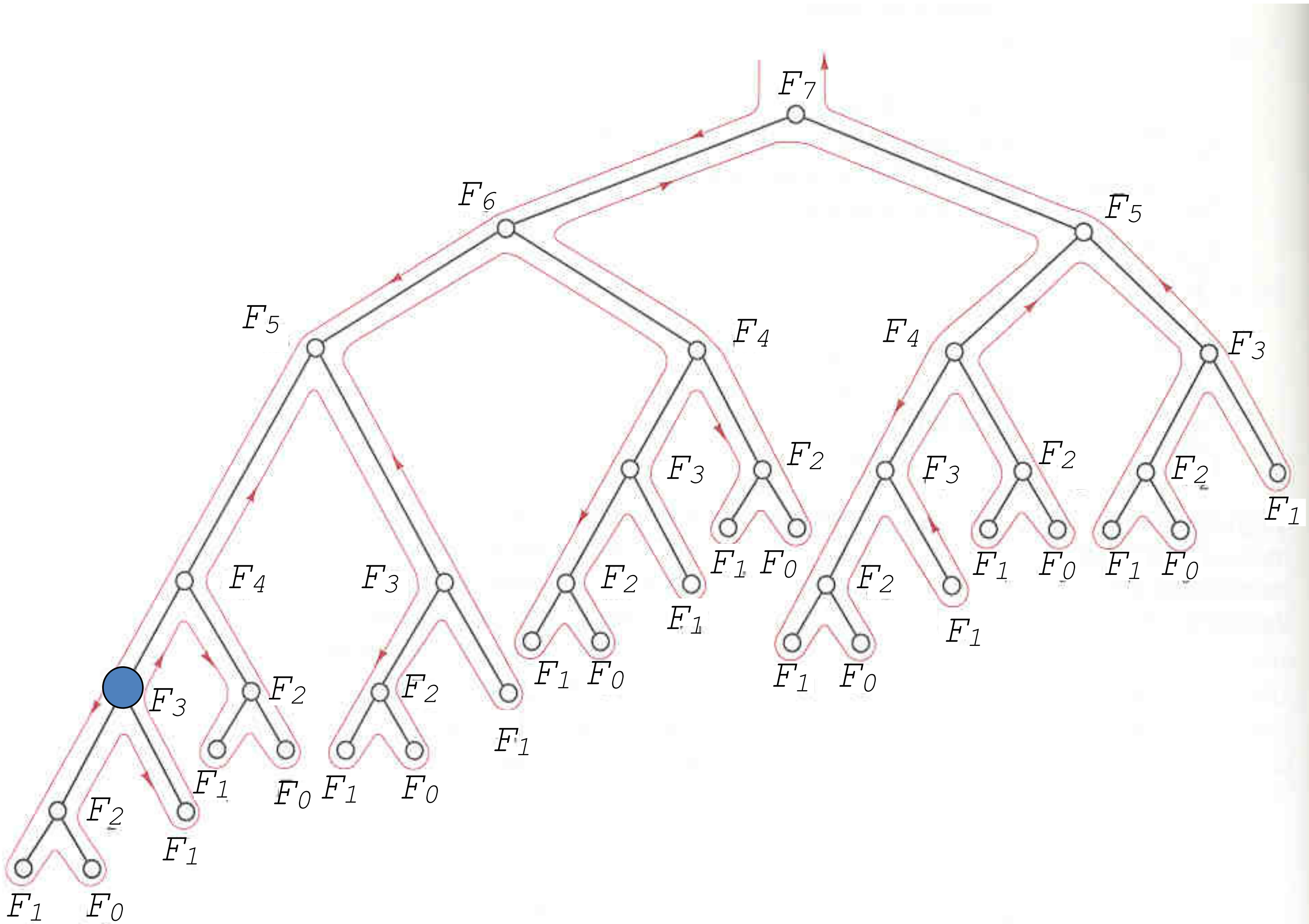


17

(Dynamic Programming)

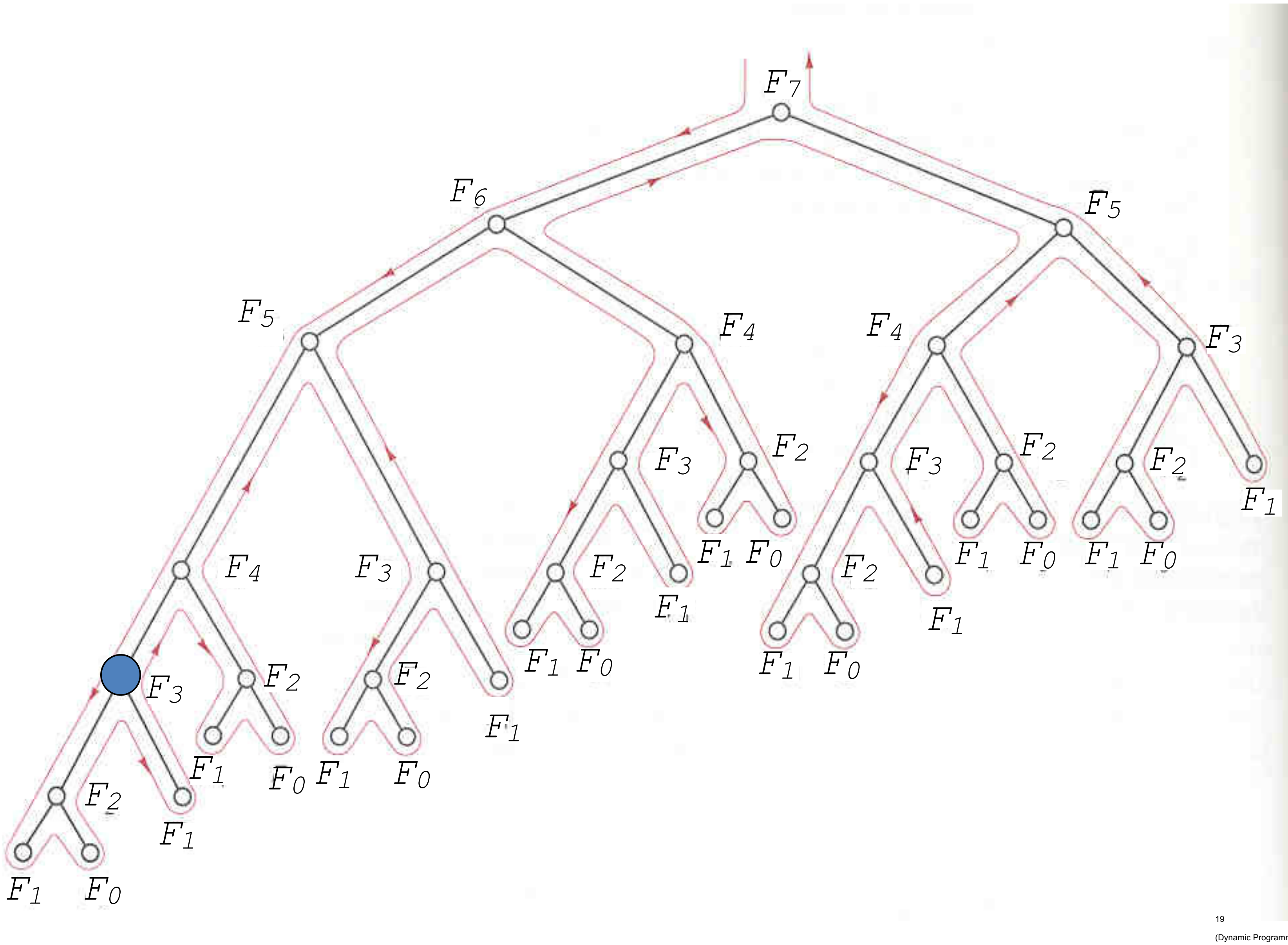
v[0]	1
v[1]	1
v[2]	2
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



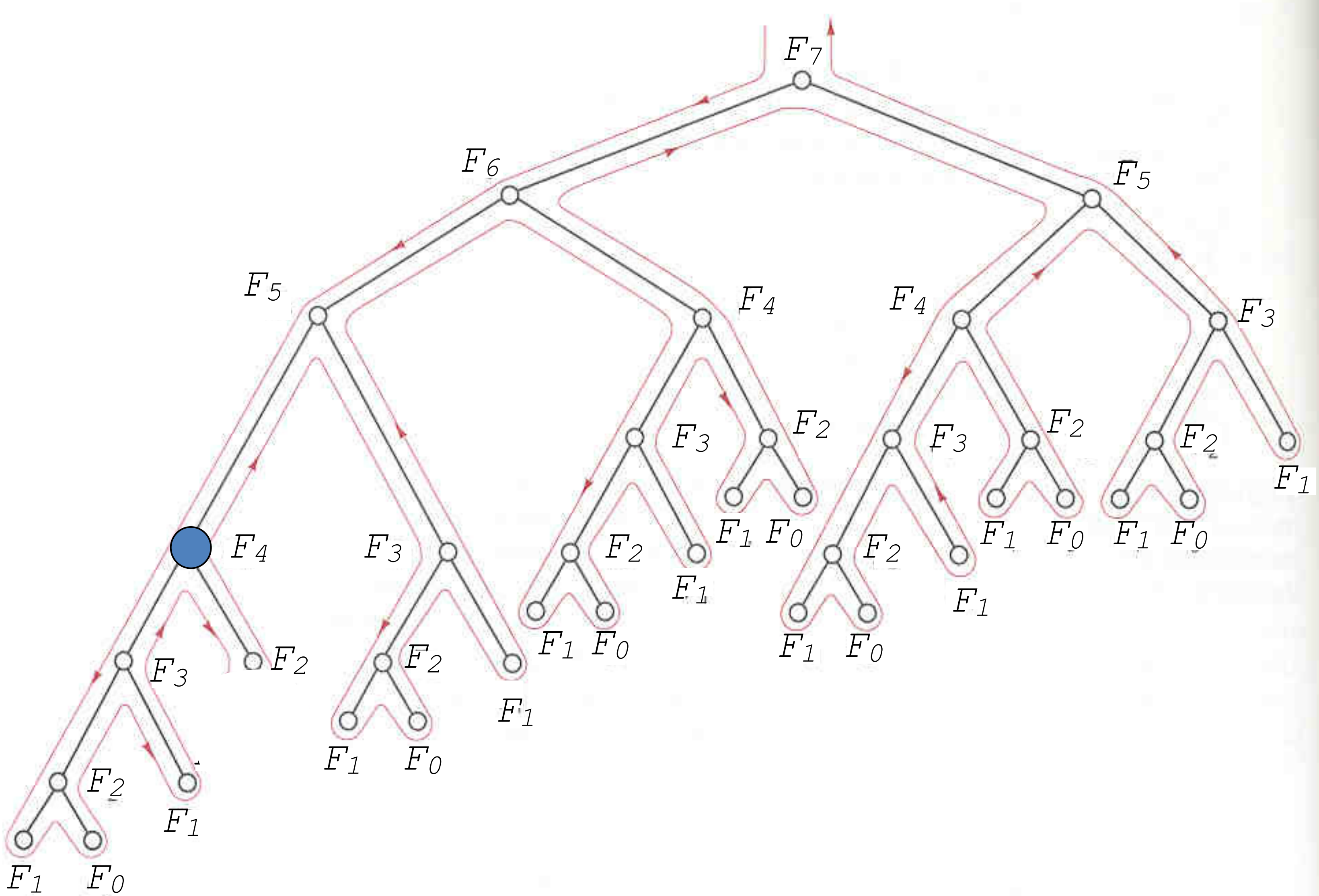
v[0]	1
v[1]	1
v[2]	2
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



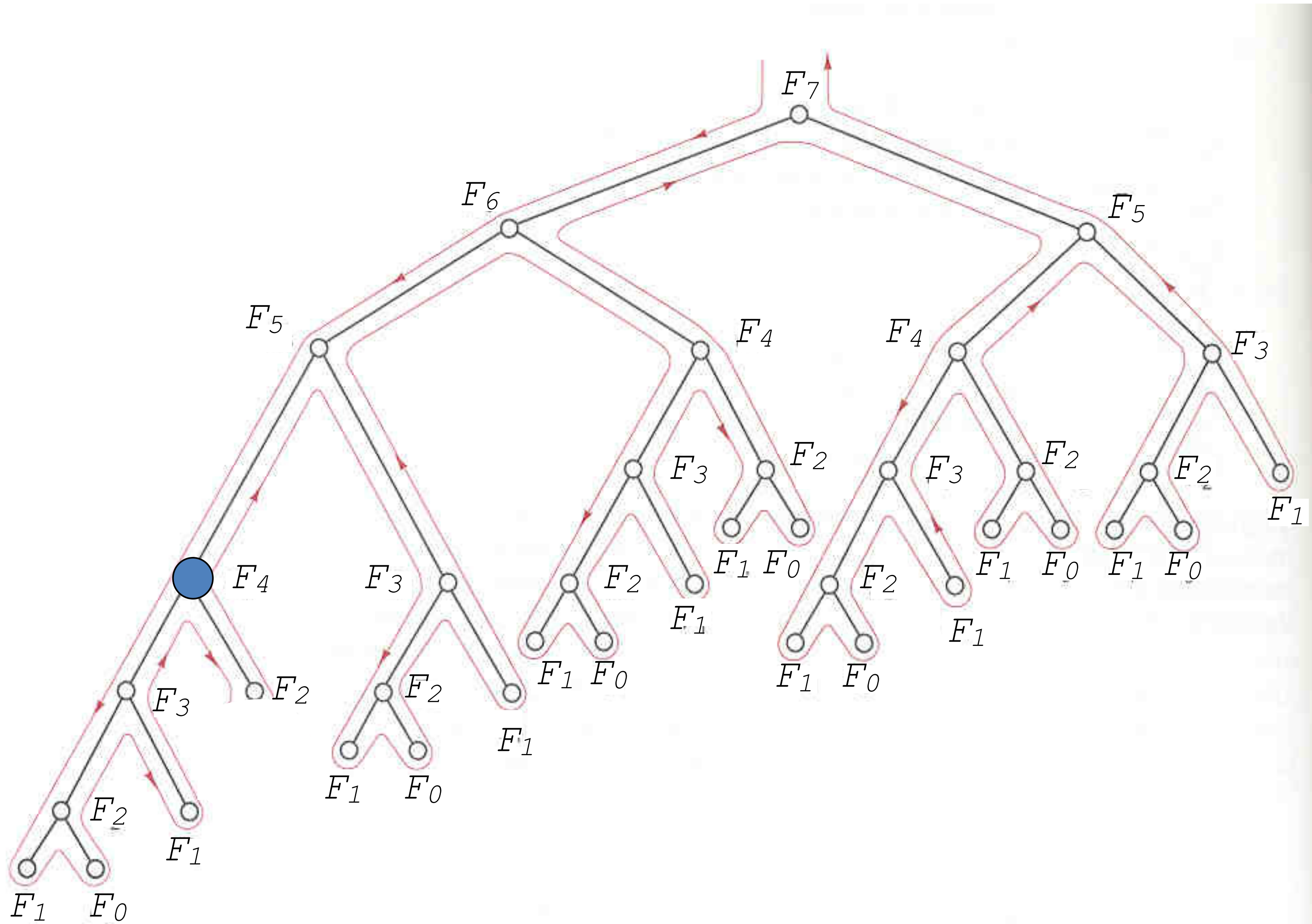
v[0]	1
v[1]	1
v[2]	2
v[3]	**3**
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



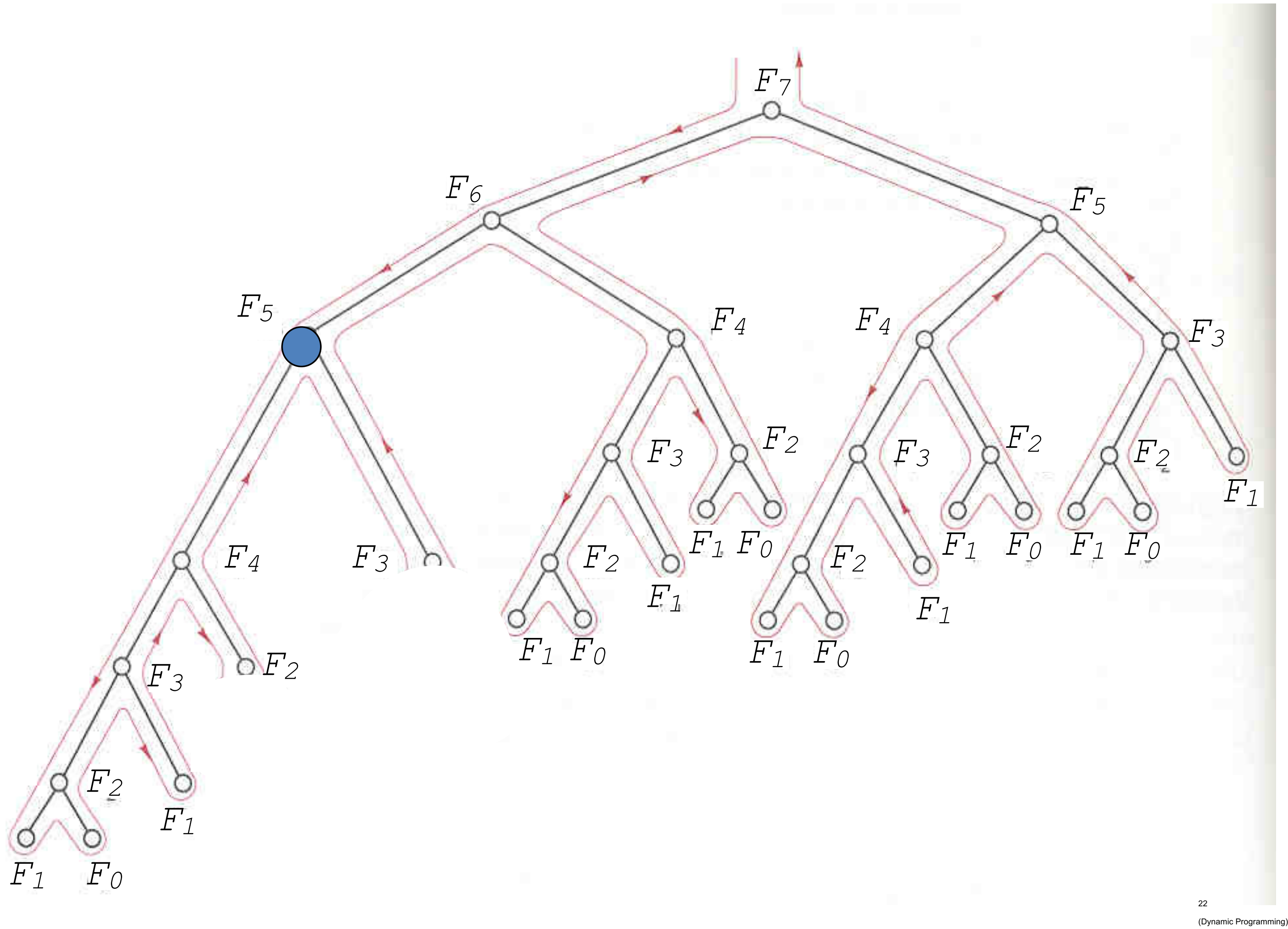
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



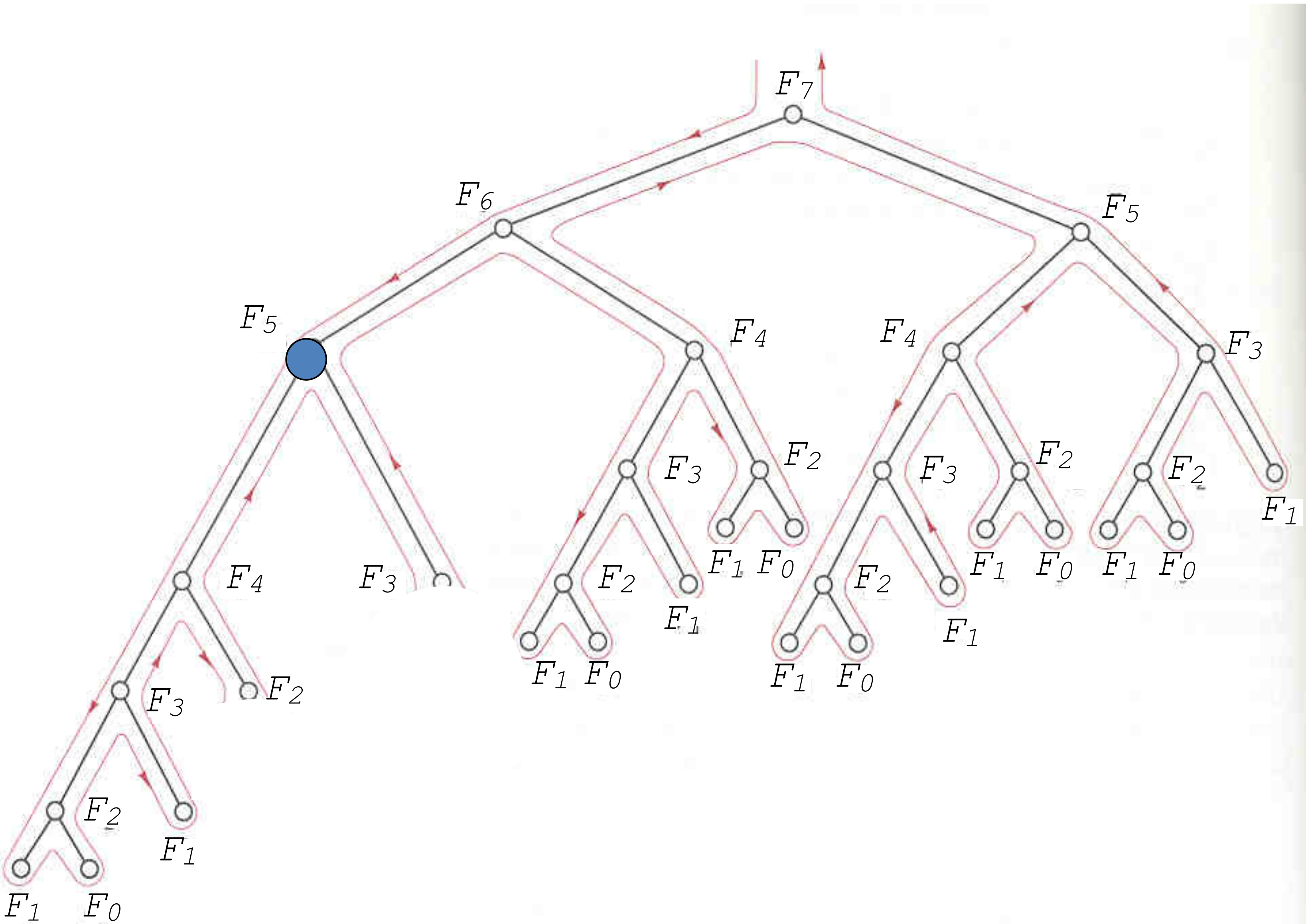
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	**5**
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



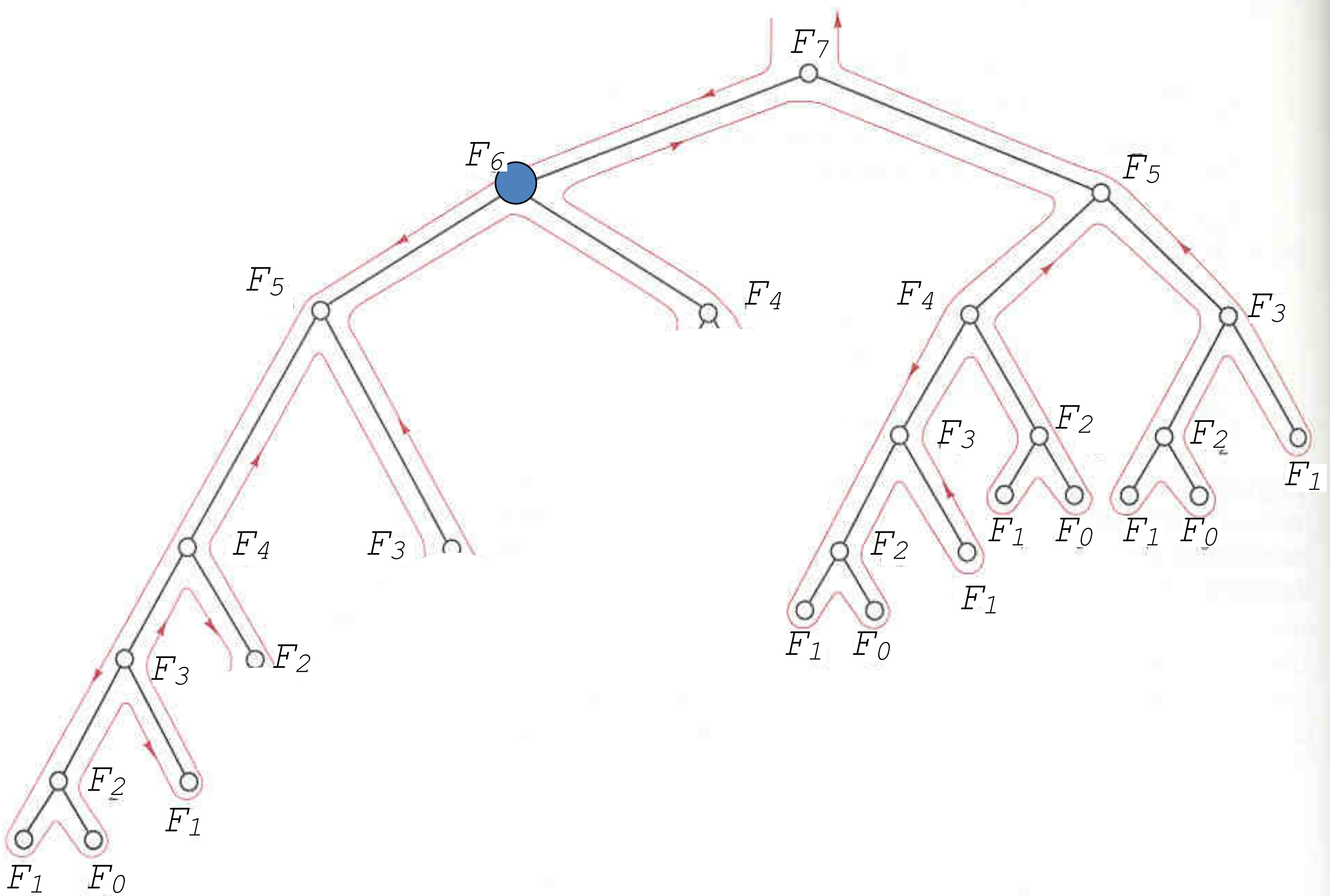
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



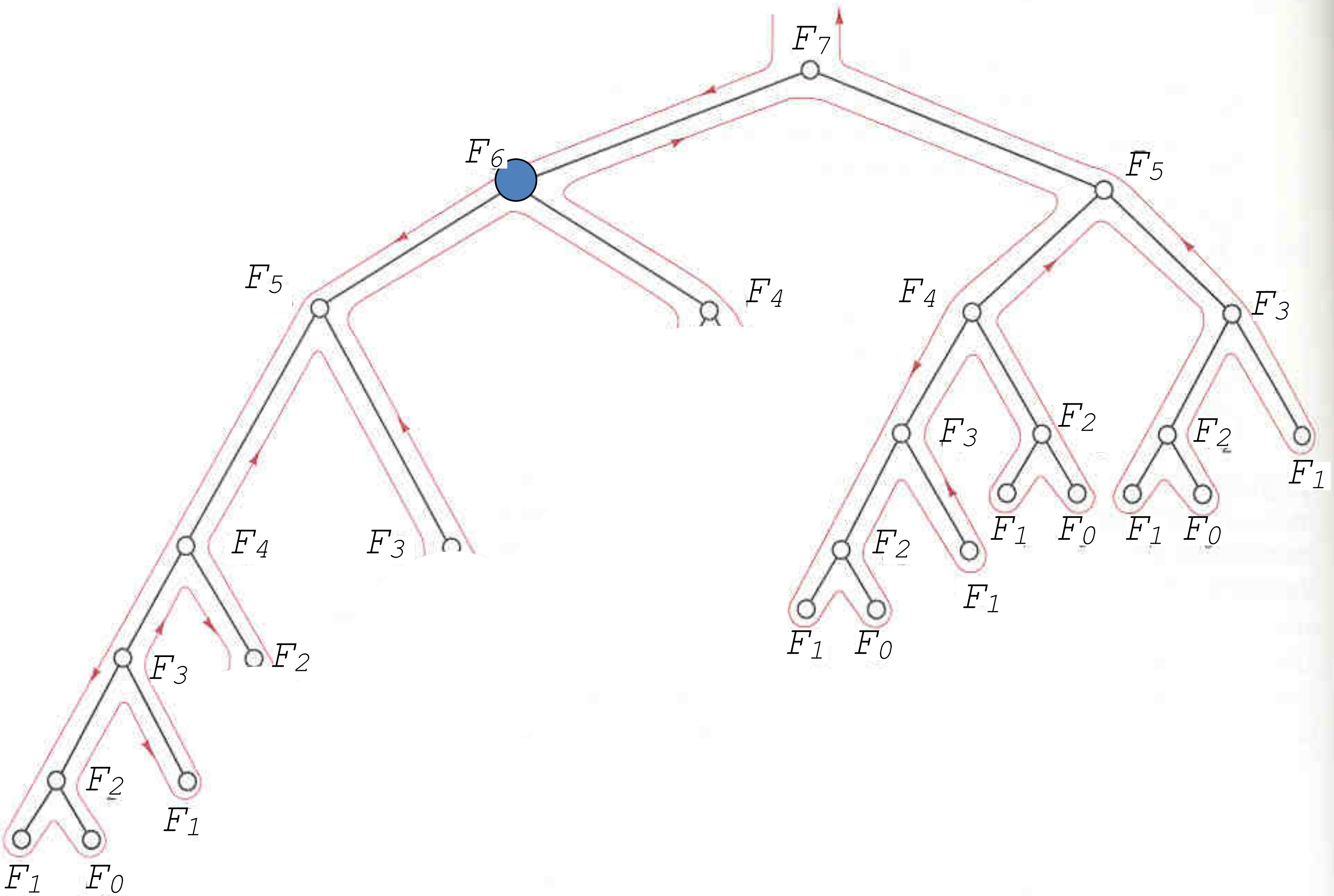
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	**8**
v[6]	-1
v[7]	-1

Look at the execution of F(7)



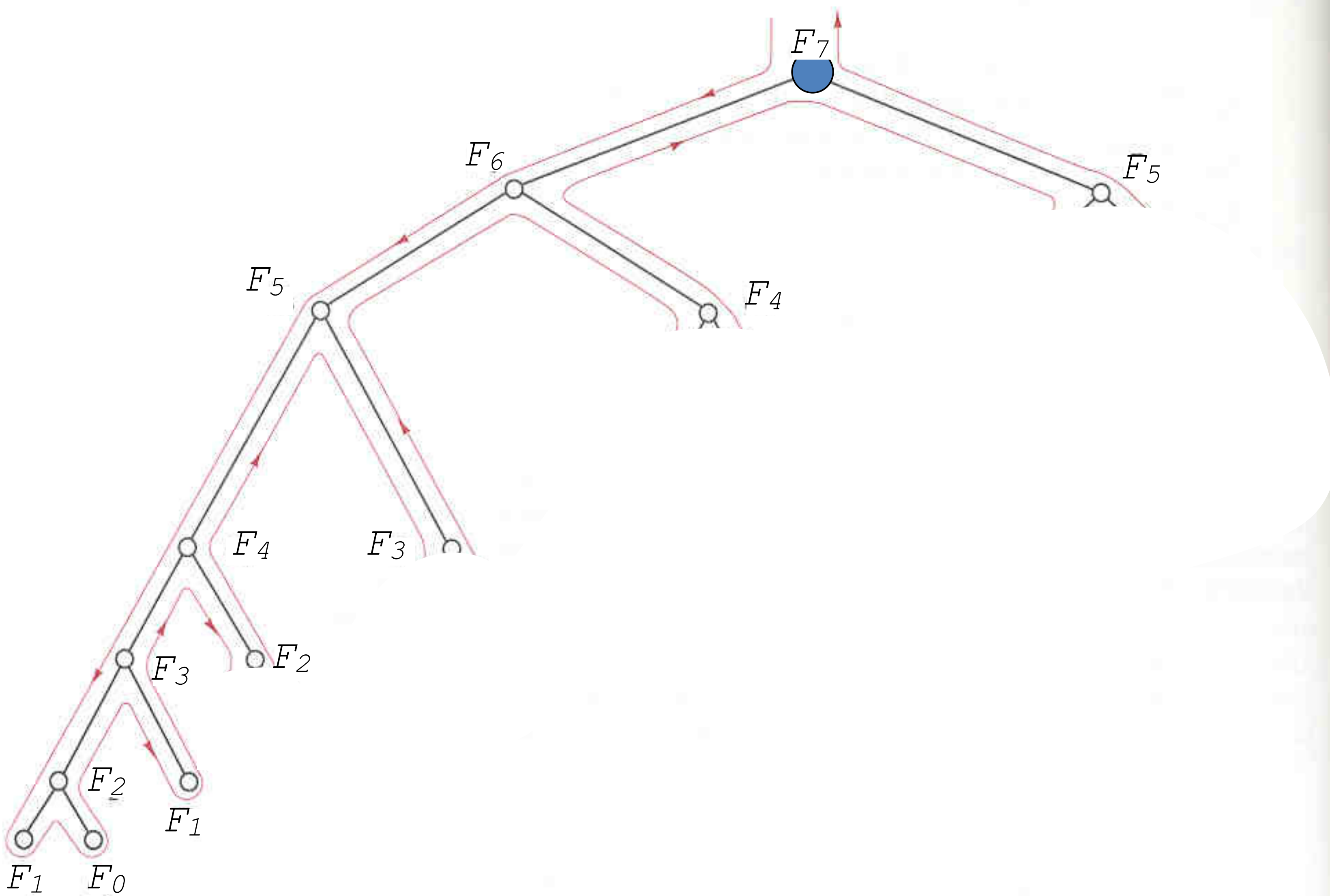
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	-1
v[7]	-1

Look at the execution of F(7)



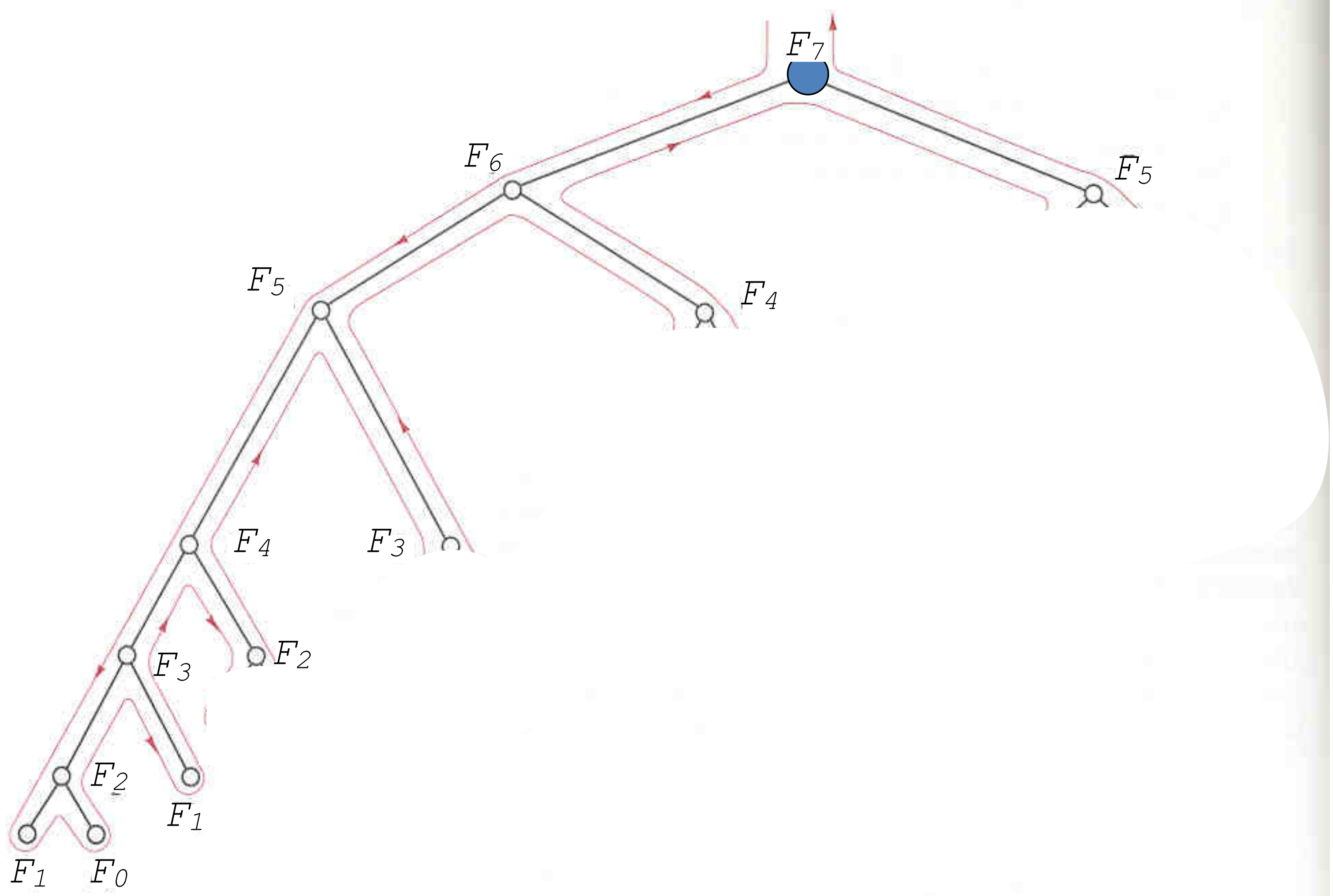
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	-1

Look at the execution of F(7)



v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	-1

Look at the execution of F(7)



v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	21

Can we do even better?

Observation

- The 2nd version stills make many function calls, and each wastes times in parameters passing, dynamic linking, ...
- In general, to compute F(i), we need F(i-1) & F(i-2) only

Idea to further improve

- Compute the values in bottom-up fashion.
- That is, compute F(2) (we already know F(0)=F(1)=1), then F(3), then F(4)…

This new implementation saves lots of overhead.

```
Procedure F(n)
    Set A[0] = A[1] = 1
    for i = 2 to n do
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

Recursive vs DP approach

Recursive version:

```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

Too Slow! exponential

Dynamic Programming version:

```
Procedure F(n)
    Set A[0] = A[1] = 1
    for i = 2 to n do
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

Efficient! Time complexity is O(n)

Summary of the methodology

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be **stored** and **retrieved** easily in a table (i.e., array)
- Fill the table in some **bottom-up** manner; start filling the solution of the smallest problem.
  - This ensures that when we solve a particular sub-problem, the solutions of all the smaller sub-problems that it depends are available.

For historical reasons, we call such methodology **Dynamic Programming**.

In the late 40's (when computers were rare), programming refers to the "tabular method".

Exercise

Consider the following function

$$G(n) = \begin{cases} 1 & \text{if } 0 \le n \le 2 \\ G(n-1) + G(n-2) + G(n-3) & \text{if } n > 2 \end{cases}$$

1. Draw the execution tree of computing **G(6)** recursively
2. Using dynamic programming, write a pseudo code to compute G(n) efficiently
3. What is the time complexity of your algorithm?

Exercise

$$G(n) = \begin{cases} 1 & \text{if } 0 \le n \le 2 \\ G(n-1) + G(n-2) + G(n-3) & \text{if } n > 2 \end{cases}$$

Dynamic Programming version:

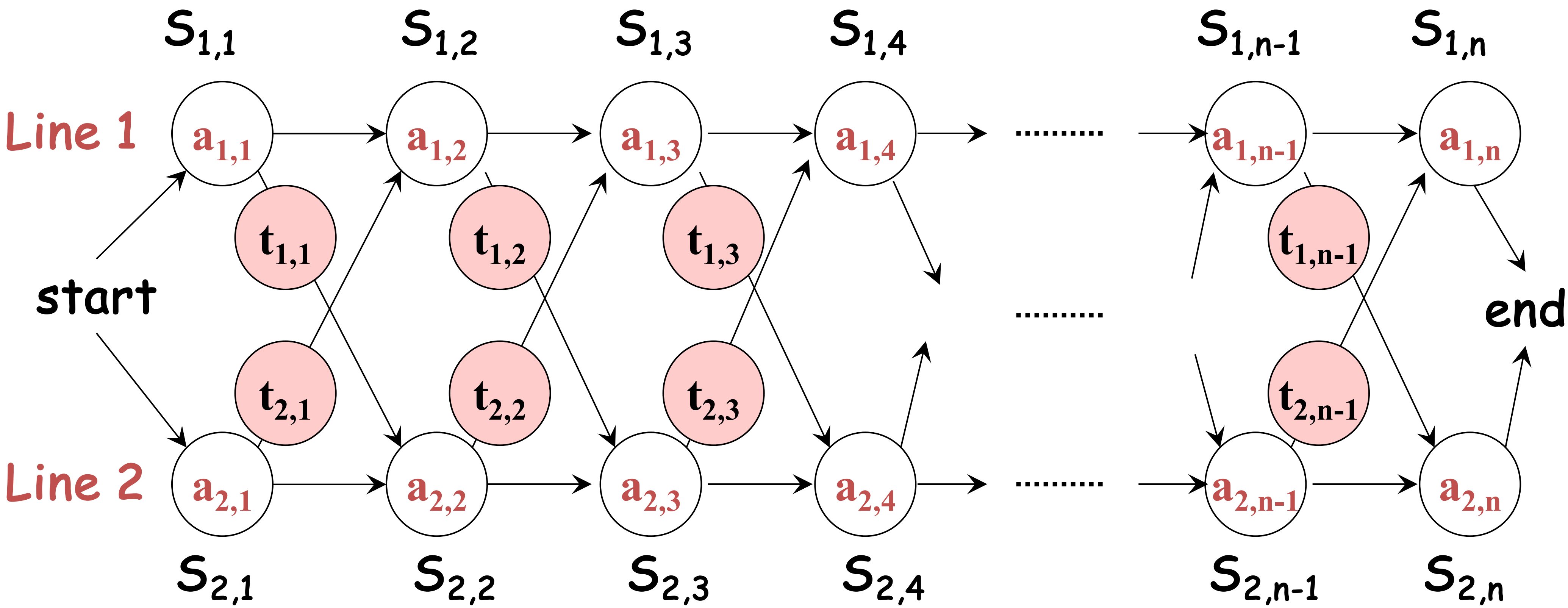
```
Procedure G(n)
   Set A[0] = A[1] = A[2] = 1
   for i = 3 to n do
      A[i] = A[i-1] + A[i-2] + A[i-3]
   return A[n]
```

Time complexity is O(n)

Assembly line scheduling

Assembly line scheduling

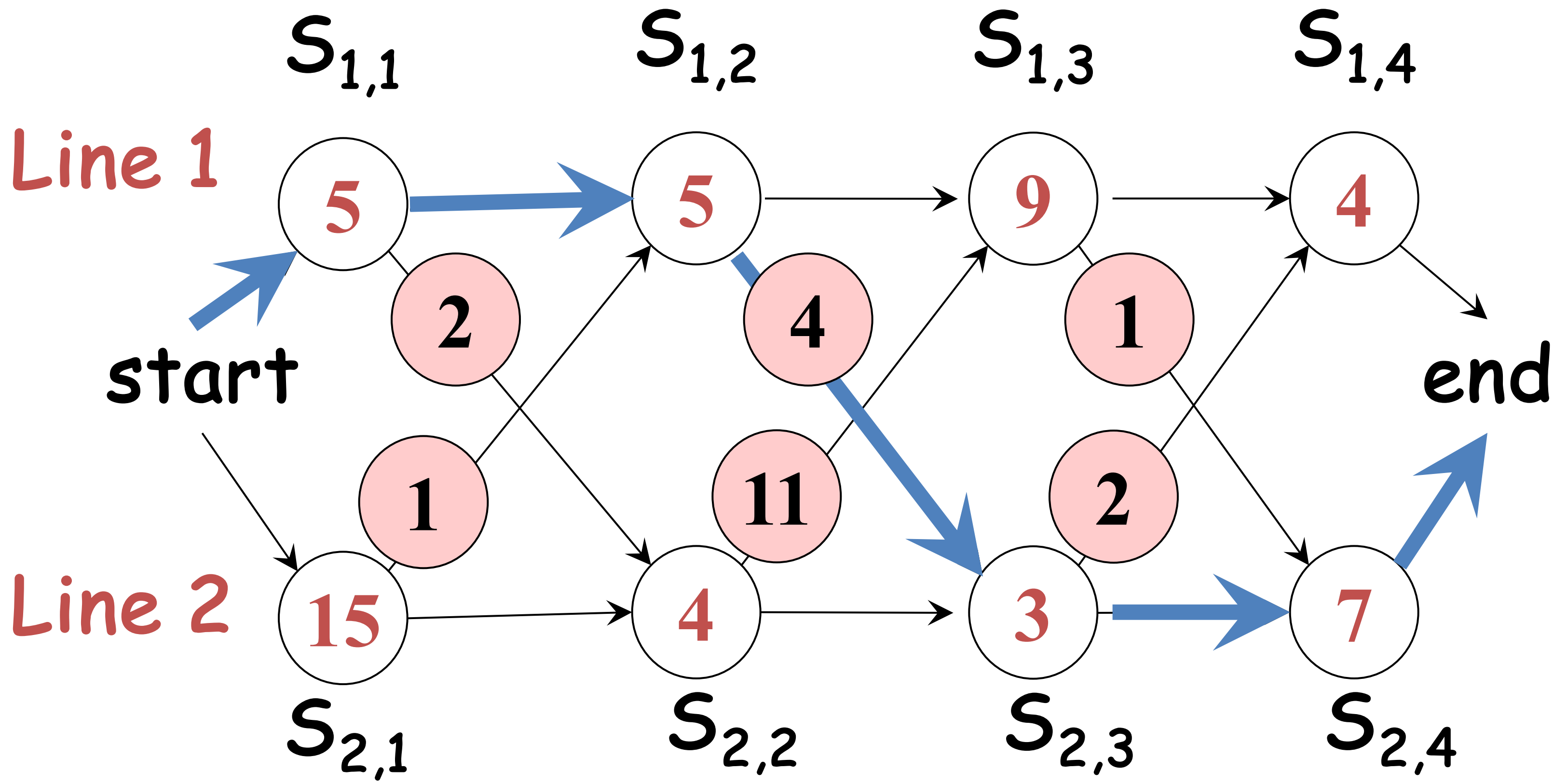
2 assembly lines, each with n stations ($S_{i,j}$: line i station j)
$S_{1,j}$ and $S_{2,j}$ perform same task but time taken is different



$a_{i,j}$: assembly time at $S_{i,j}$
$t_{i,j}$: transfer time after $S_{i,j}$

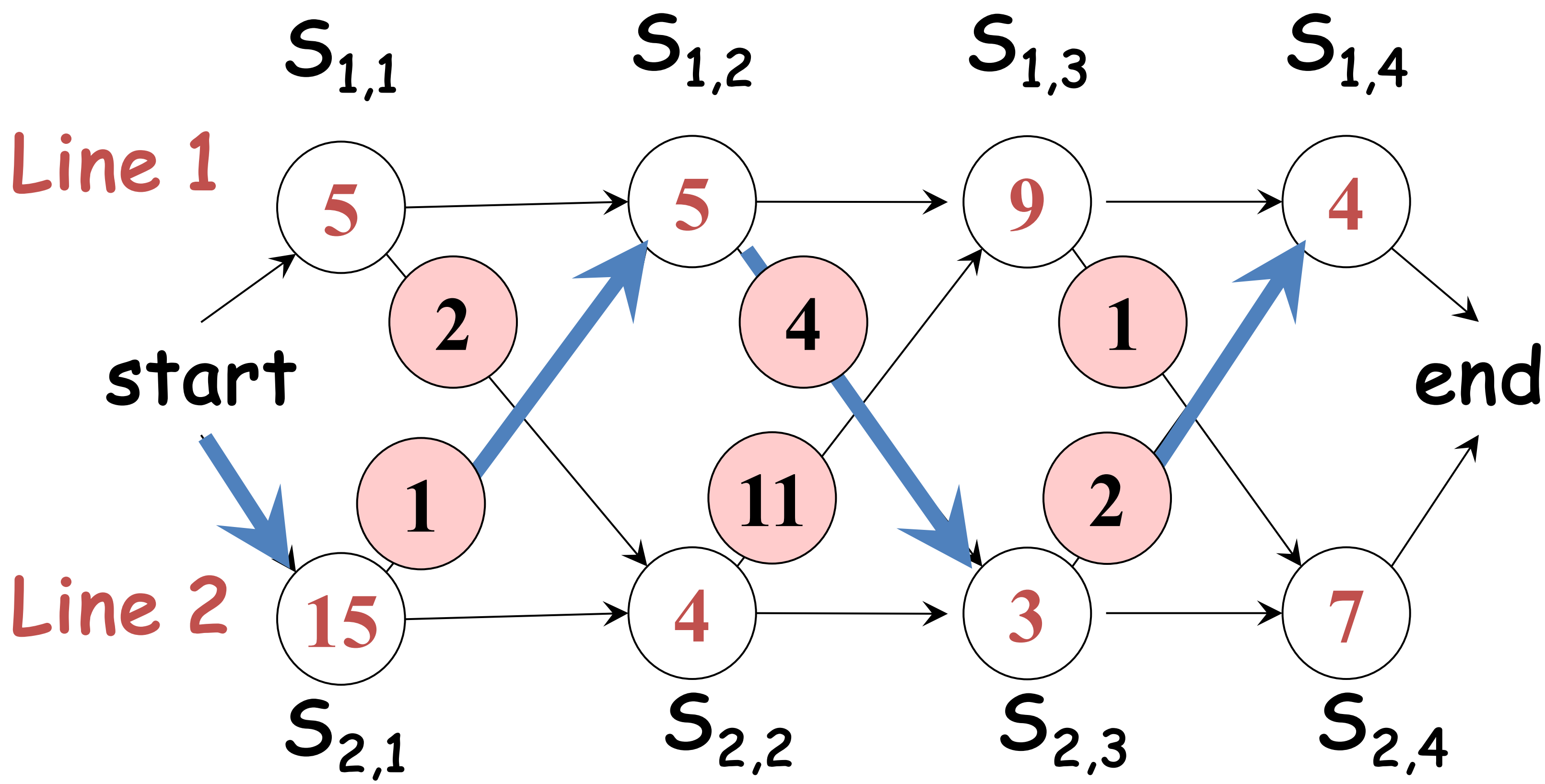
Problem: To determine which stations to go in order to **minimize** the total time through the n stations

Example (1)



stations chosen:	$S_{1,1}$	$S_{1,2}$		$S_{2,3}$	$S_{2,4}$	
time required:	5	5	4	3	7	= 24

Example (2)



stations chosen:	$S_{1,1}$		$S_{1,2}$		$S_{2,3}$		$S_{2,4}$	
time required:	5		5	4	3		7	= 24
stations chosen:	$\mathbf{S_{2,1}}$		$\mathbf{S_{1,2}}$		$\mathbf{S_{2,3}}$		$\mathbf{S_{1,4}}$	
time required:	**15**	**1**	**5**	**4**	**3**	**2**	**4**	**= 34**

Example

$S_{1,1}$

Line 1

How to determine the best stations to go?
There are altogether 2^n choices of stations.
Should we try them all?

stations chos

time required: 5 7 = 24

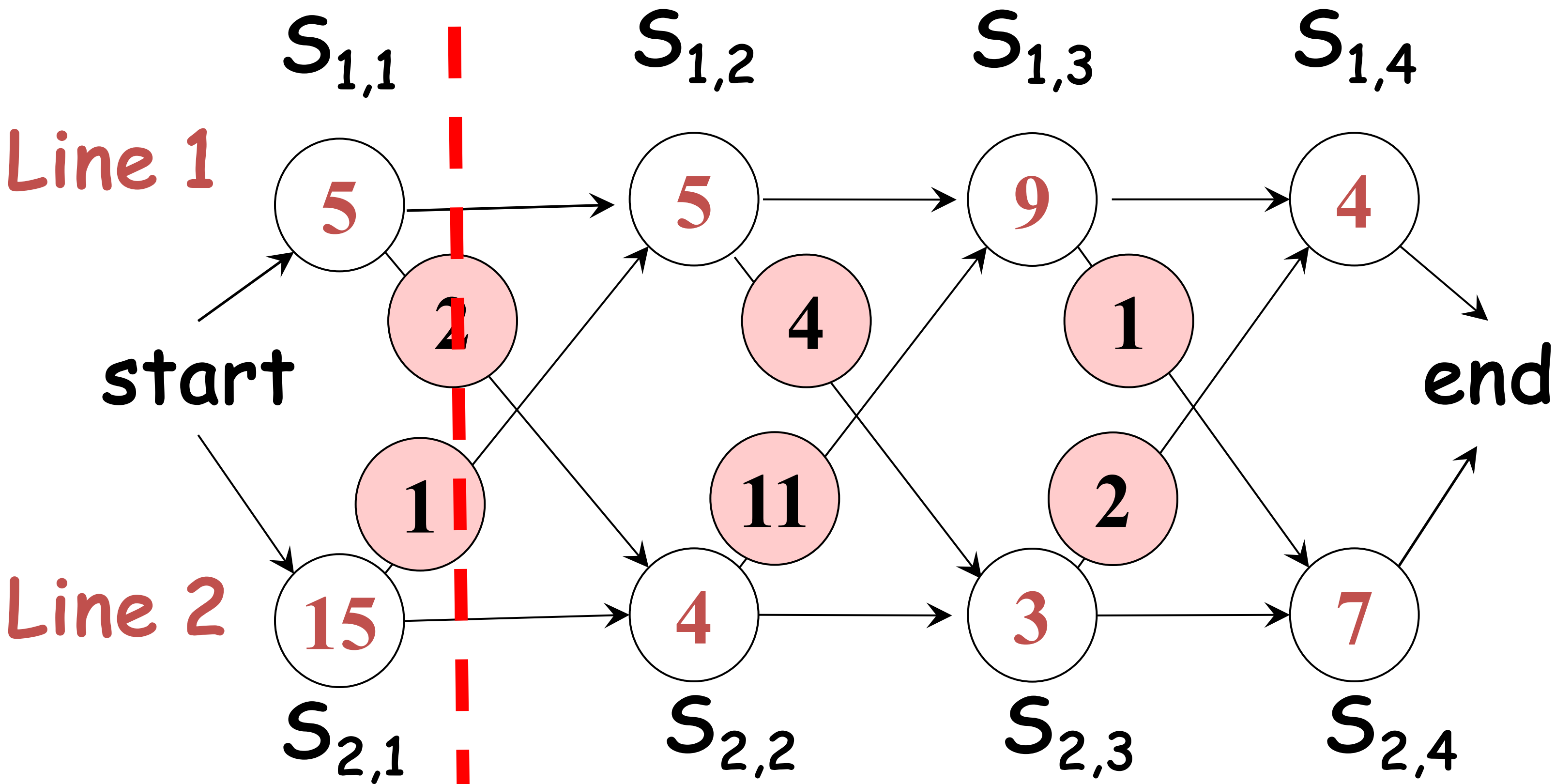
stations chosen: $S_{1,2}$ $S_{2,3}$ $S_{1,4}$

time required: 15 1 5 4 3 2 4 = 34

Good news: Dynamic Programming

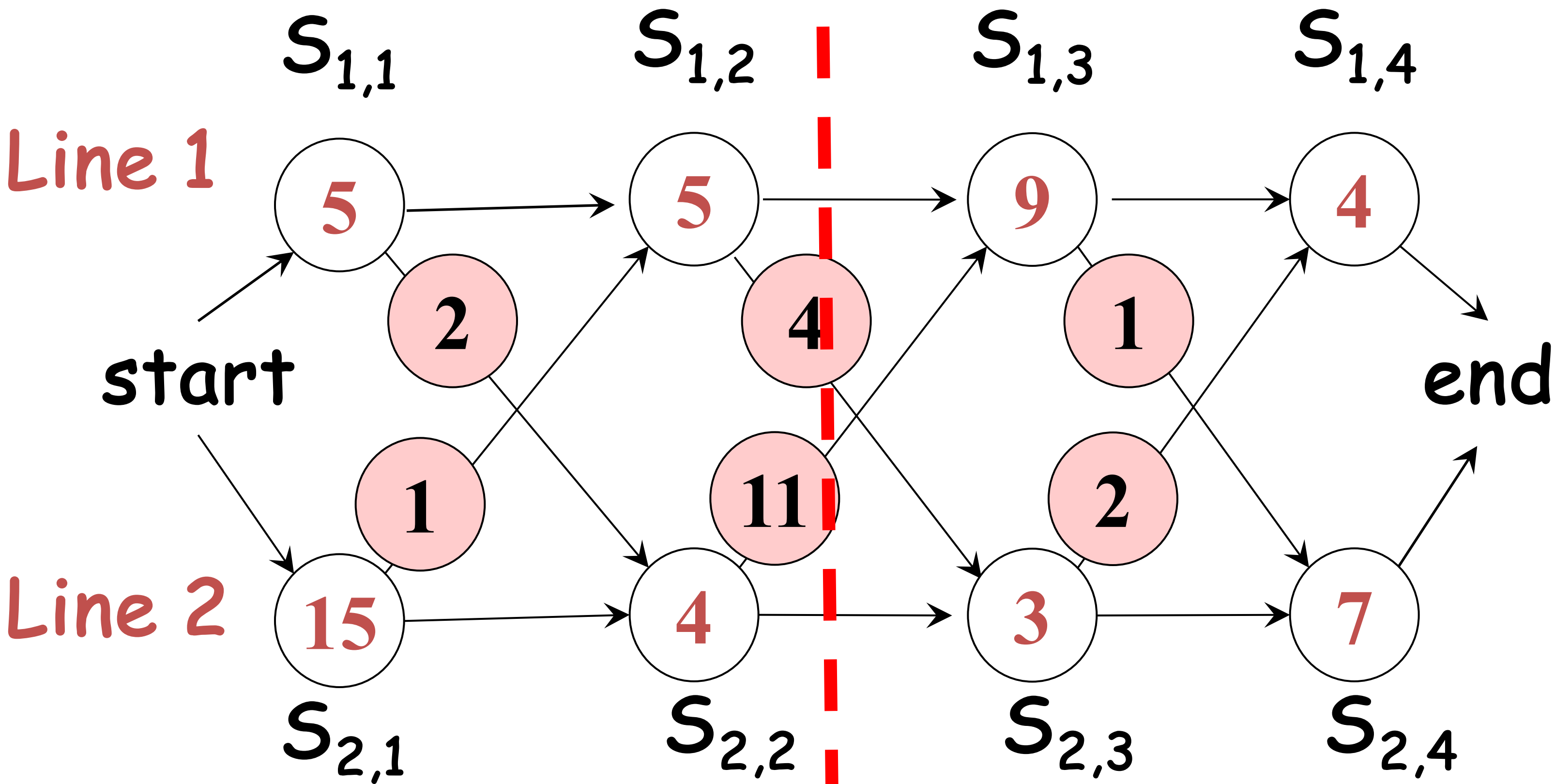
- We **don't** need to try all possible choices.
- We can make use of **dynamic programming**:
  1. If we can compute the fastest ways to get thro' station $\mathbf{S_{1,n}}$ and $\mathbf{S_{2,n}}$, then the faster of these two ways is the overall fastest way.
  2. To compute the fastest ways to get thro' $S_{1,n}$ (similarly for $S_{2,n}$), we need to know the fastest way to get thro' $\mathbf{S_{1,n-1}}$ and $\mathbf{S_{2,n-1}}$
  3. In general, we want to know the fastest way to get thro' $\mathbf{S_{1,j}}$ and $\mathbf{S_{2,j}}$, for all j.

Example again



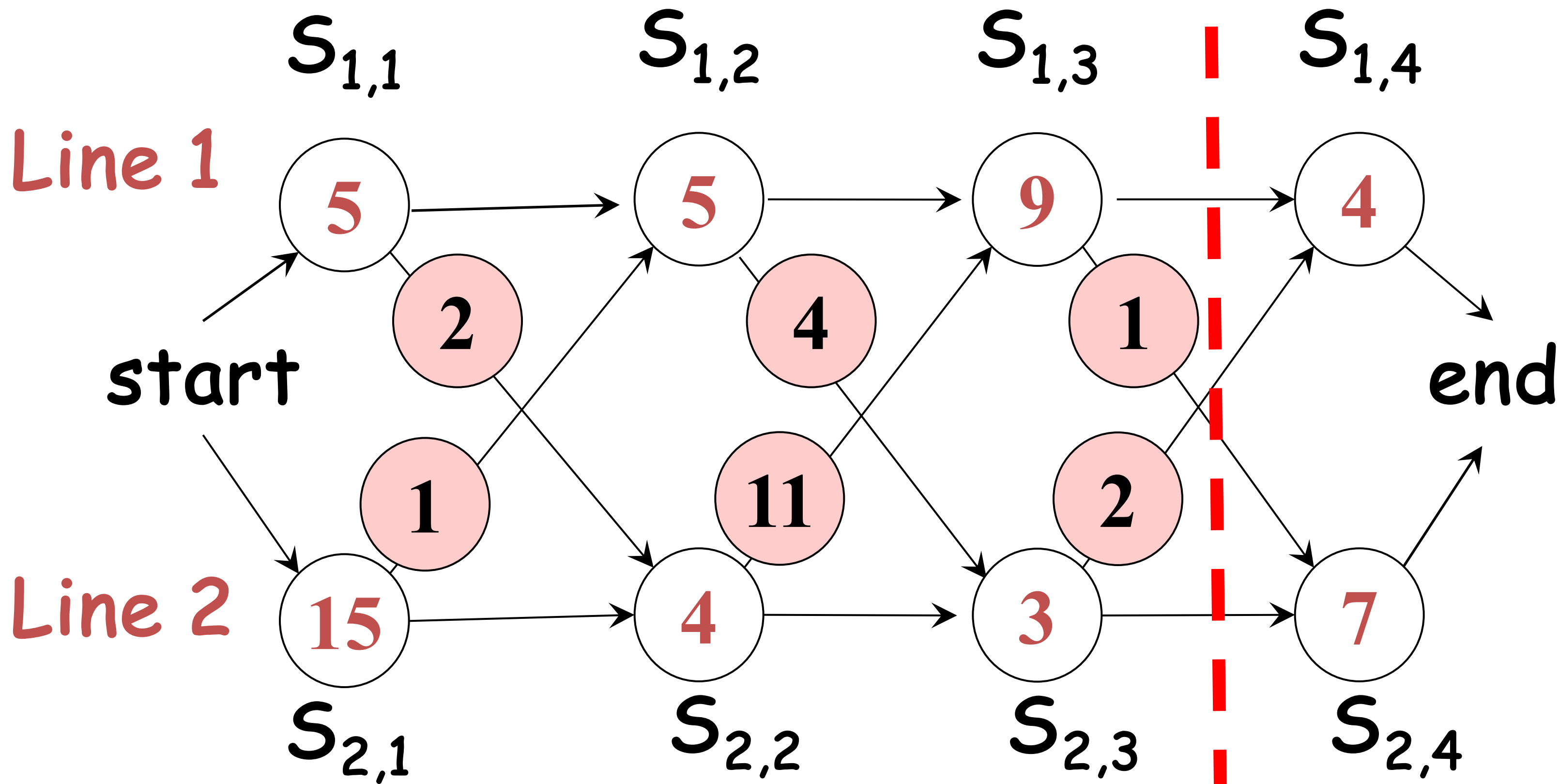
	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	S_{14}
minimum cost:	5			
	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
	15			

Example again



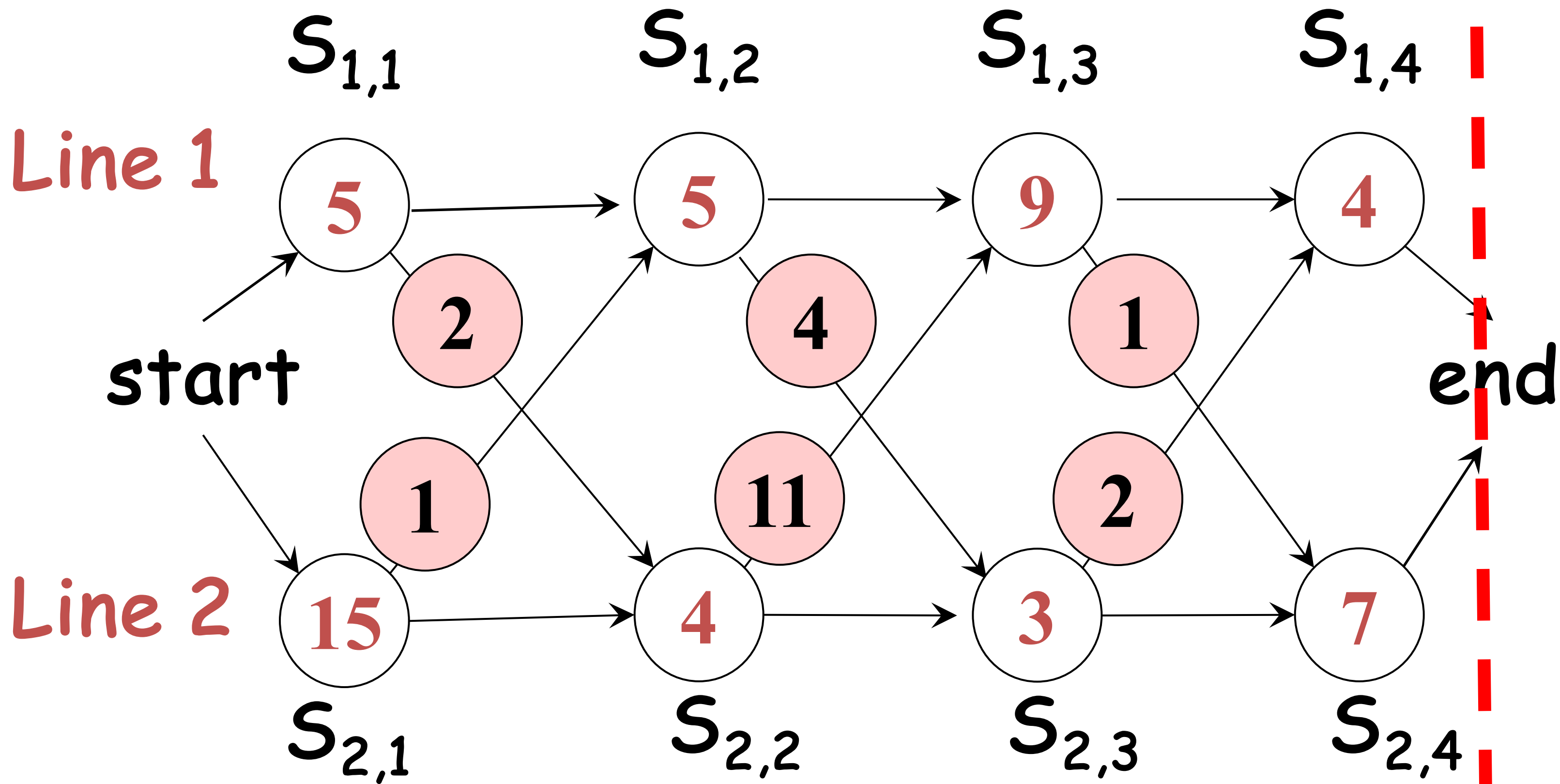
	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	S_{14}
minimum cost:	5	5+5 ~~15+1+5~~		
	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
	15	5+2+4 ~~15+4~~		

Example again



	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	S_{14}
minimum cost:	5	5+5 ~~15+1+5~~	10+9 ~~11+11+9~~	
	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
	15	5+2+4 ~~15+4~~	~~10+4+3~~ 11+3	

Example again



	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	S_{14}	
		5+5	10+9	~~19+4~~	
minimum cost:	5	~~15+1+5~~	~~11+11+9~~	14+2+4	20
	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	
	15	5+2+4	~~10+4+3~~	~~19+1+7~~	~~21~~
		~~15+4~~	11+3	14+7	

A dynamic programming solution

What are the sub-problems?

- given j, what is the fastest way to get thro' $S_{1,j}$
- given j, what is the fastest way to get thro' $S_{2,j}$

Definitions:

- **$f_1[j]$** = the fastest time to get thro' $S_{1,j}$
- **$f_2[j]$** = the fastest time to get thro' $S_{2,j}$

The final solution equals to **min { $f_1[n]$, $f_2[n]$ }**

Task:

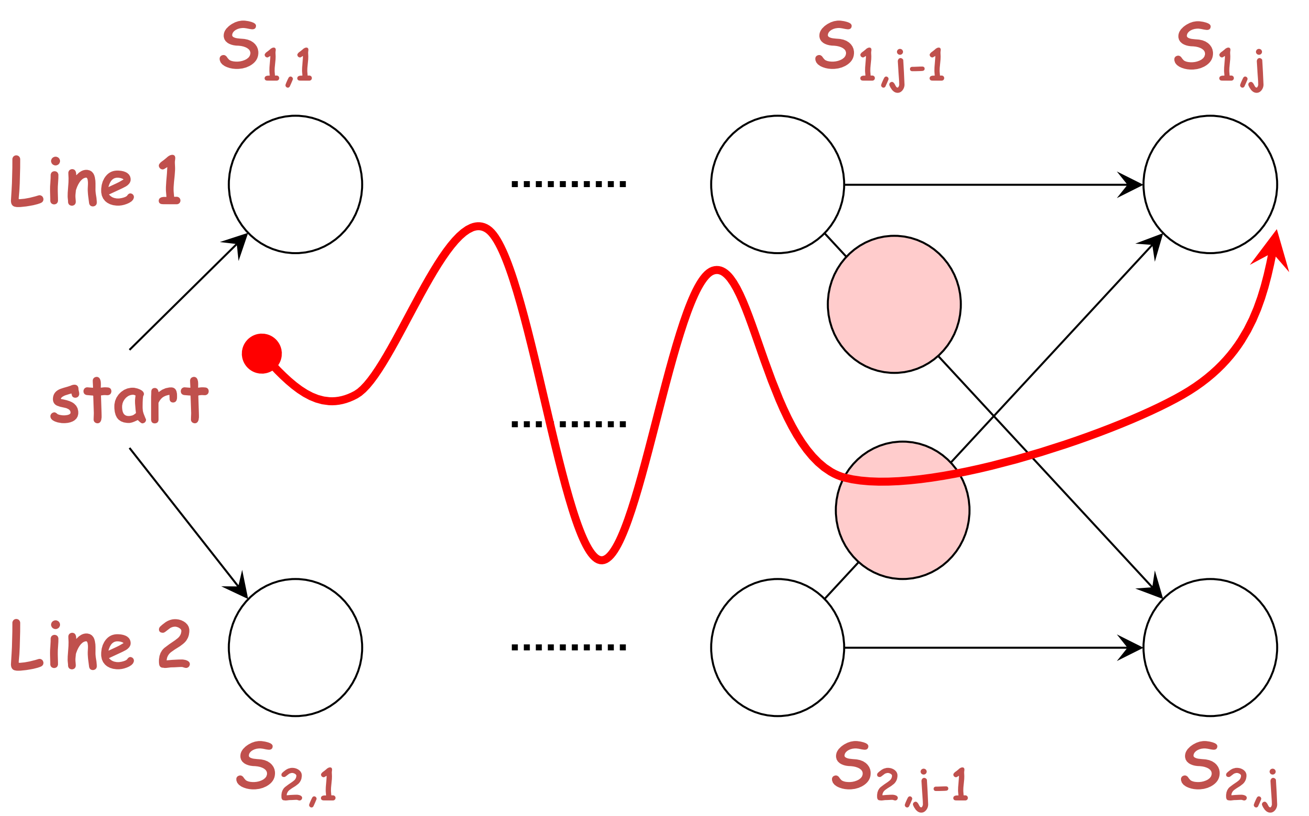
- Starting from $f_1[1]$ and $f_2[1]$,
  compute $f_1[j]$ and $f_2[j]$ incrementally

Solving the sub-problems (1)

Q1: what is the fastest way to get thro' $S_{1,j}$?

A: either

- the fastest way thro' $S_{1,j-1}$, then **directly** to $S_{1,j}$, or
- the fastest way thro' $S_{2,j-1}$, a **transfer** from line 2 to line 1, and then through $S_{1,j}$

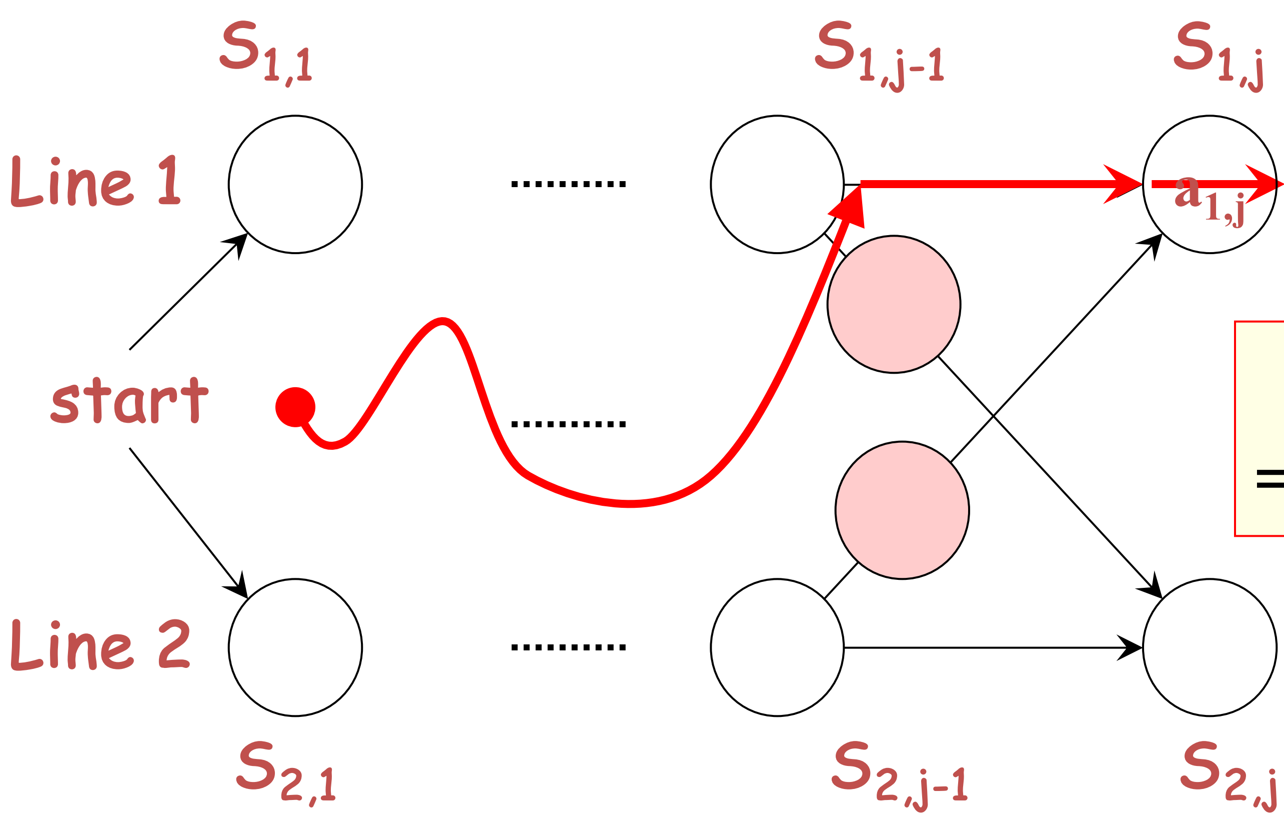


Solving the sub-problems (1)

Q1: what is the fastest way to get thro' $S_{1,j}$?

A: either

- ***the fastest way thro' $S_{1,j-1}$, then directly to $S_{1,j}$,*** or
- the fastest way thro' $S_{2,j-1}$, a transfer from line 2 to line 1, and then through $S_{1,j}$



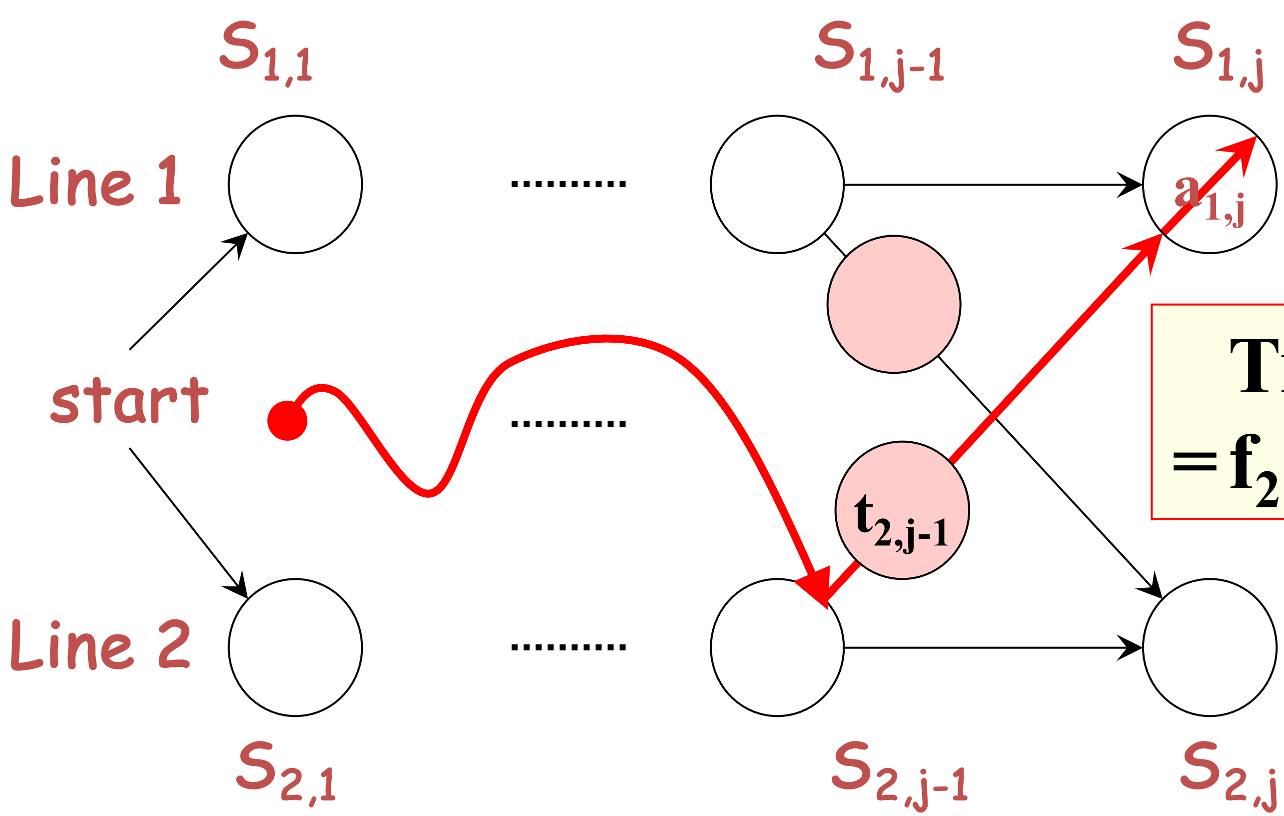
Time required
$$= f_1[j-1] + a_{1,j}$$

Solving the sub-problems (1)

Q1: what is the fastest way to get thro' $S_{1,j}$?

A: either

- the fastest way thro' $S_{1,j-1}$, then directly to $S_{1,j}$, or
- ***the fastest way thro' $S_{2,j-1}$, a transfer from line 2 to line 1, and then through $S_{1,j}$***



Time required
$$= f_2[j-1] + t_{2,j-1} + a_{1,j}$$

Solving the sub-problems (1)

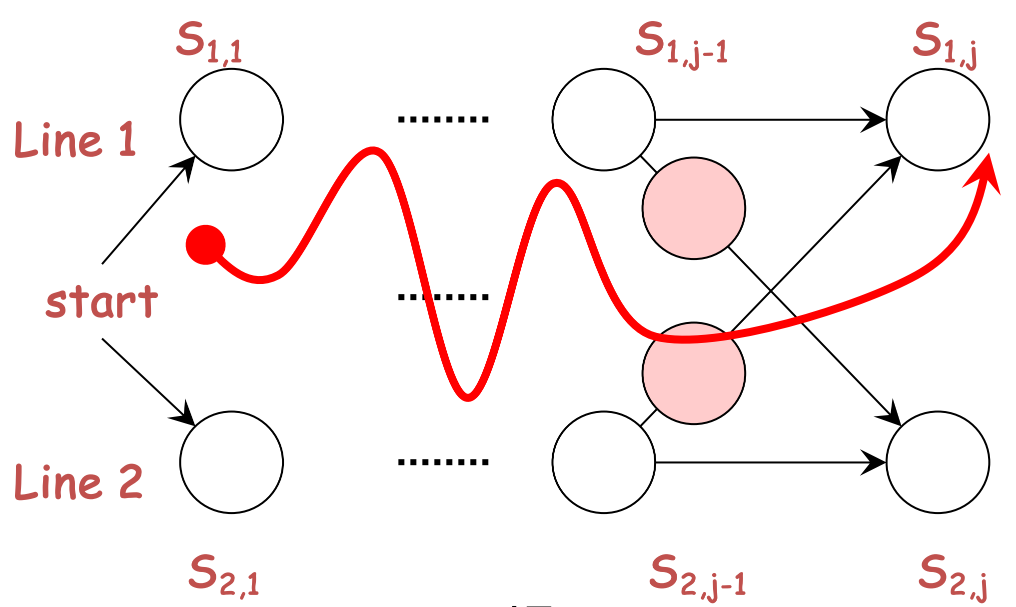
Q1: what is the fastest way to get thro' $S_{1,j}$?

A: either

- the fastest way thro' $S_{1,j-1}$, then directly to $S_{1,j}$, or
- the fastest way thro' $S_{2,j-1}$, a transfer from line 2 to line 1, and then through $S_{1,j}$

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

Boundary case: $$f_1[1] = a_{1,1}$$



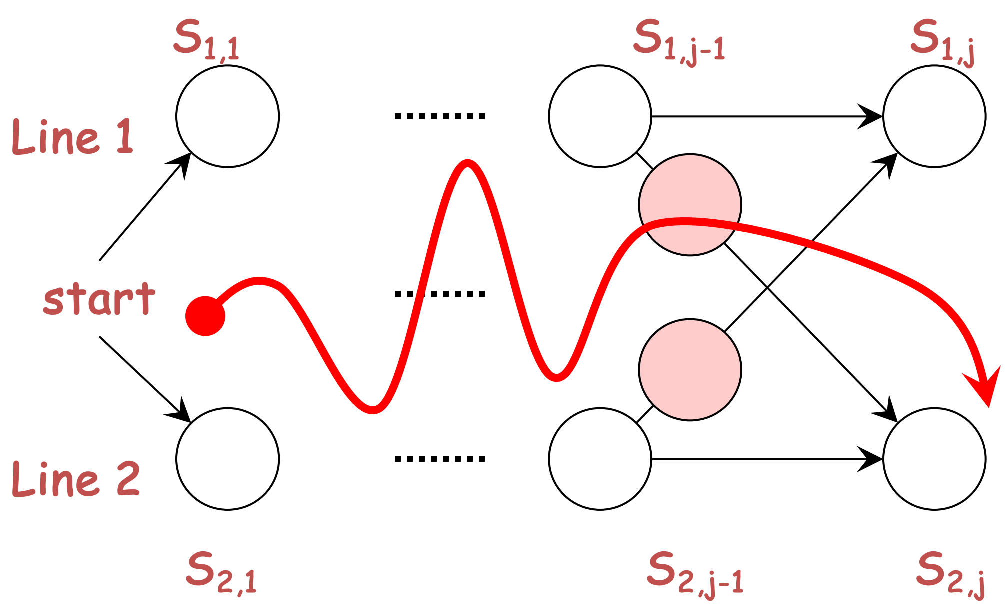
Solving the sub-problems (2)

Q2: what is the fastest way to get thro' $S_{2,j}$?

By exactly the same analysis, we obtain the formula for the fastest way to get thro' $S_{2,j}$:

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

Boundary case: $\boxed{f_2[1] = a_{2,1}}$

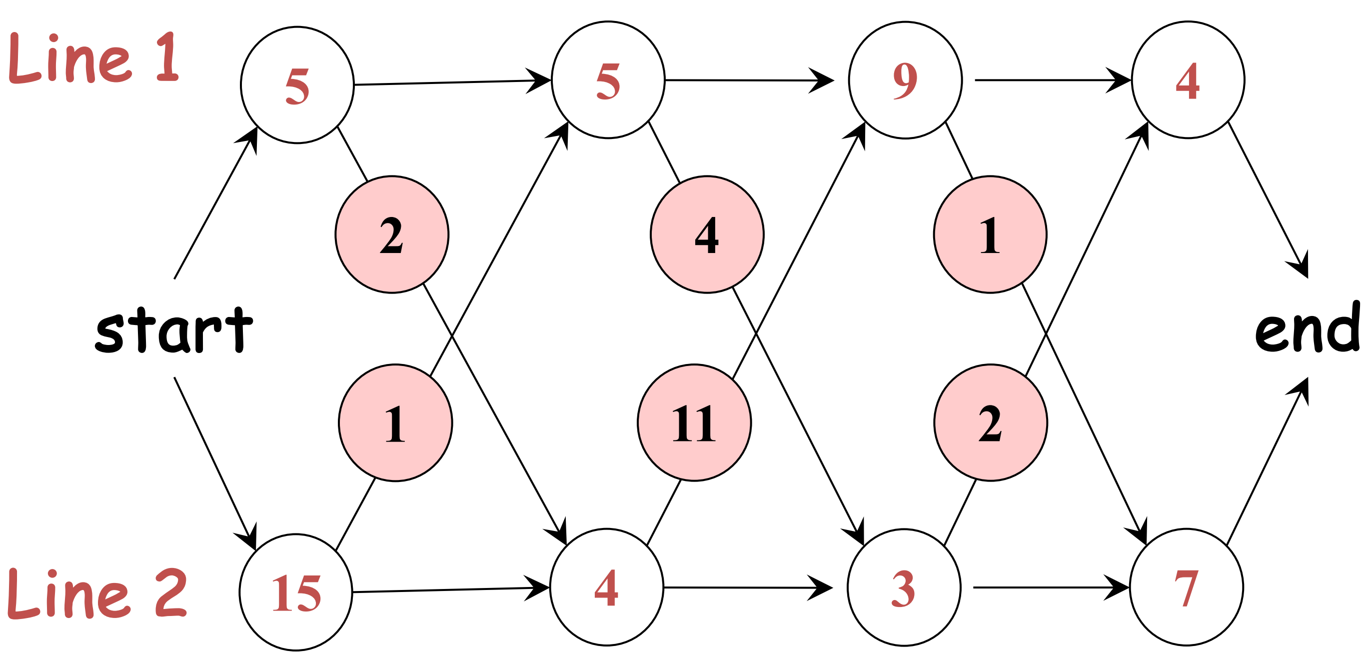


Summary

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

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

$$f^* = \min(\, f_1[n] \,,\, f_2[n]\,)$$



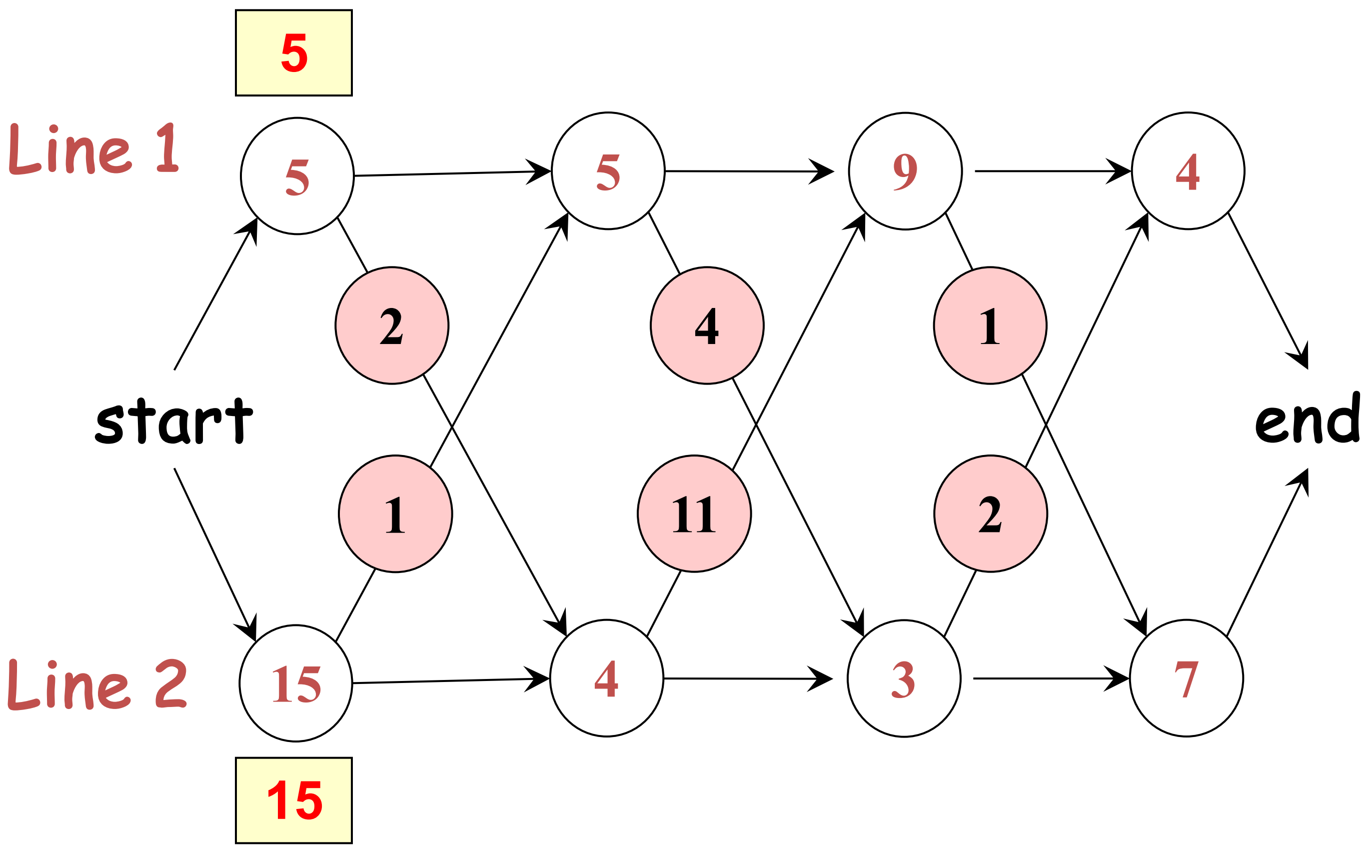
j	$f_1[j]$	$f_2[j]$
1		
2		
3		
4		

Summary

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

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

$$f^* = \min(\, f_1[n] \,,\, f_2[n] \,)$$



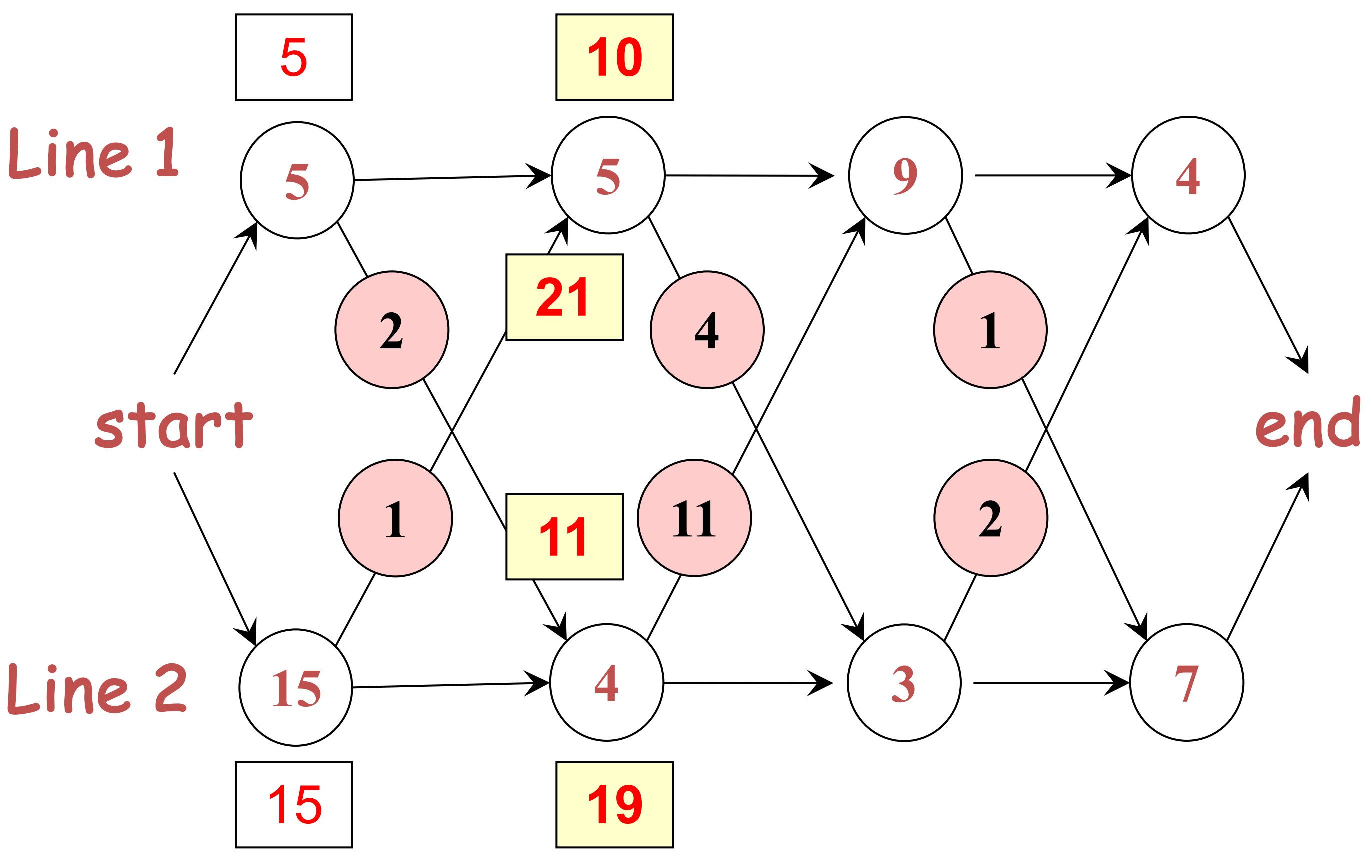
j	$f_1[j]$	$f_2[j]$
1	5	15
2		
3		
4		

Summary

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

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

$$f^* = \min(\, f_1[n] \,,\, f_2[n] \,)$$



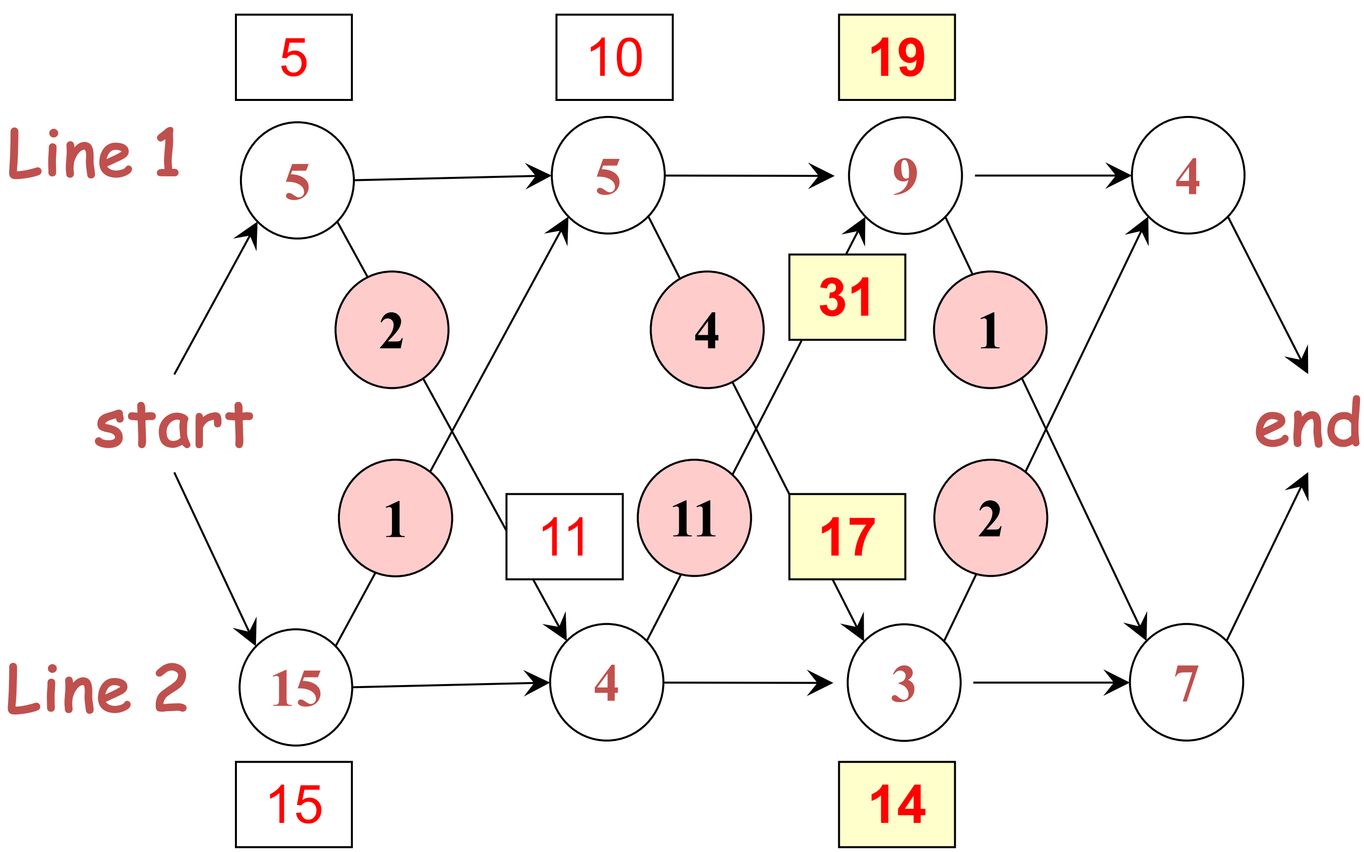
j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3		
4		

Summary

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

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

$$f^* = \min(\, f_1[n] \,,\, f_2[n] \,)$$



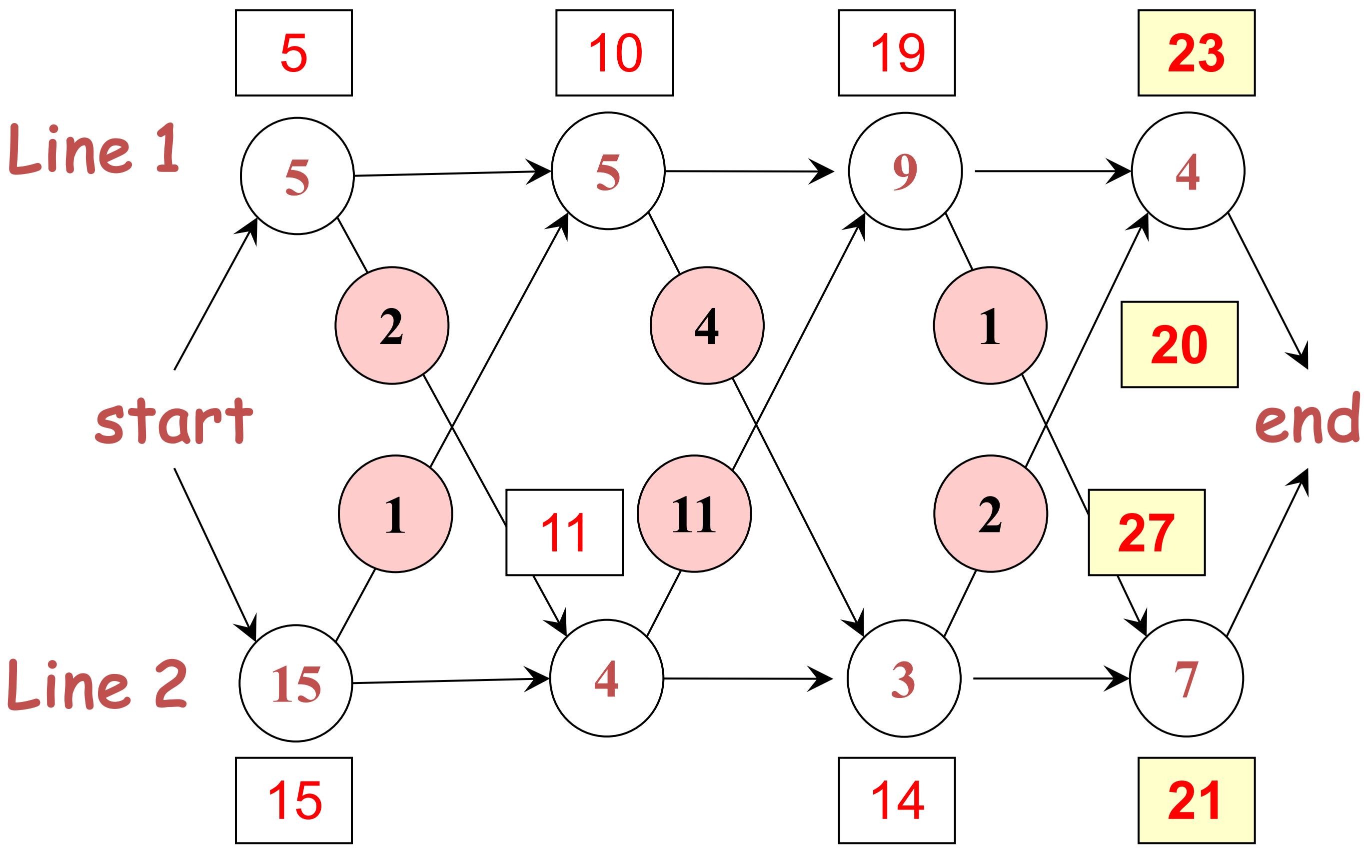
j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3	19	14
4		

Summary

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

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

$$f^* = \min(f_1[n],\ f_2[n])$$



j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3	19	14
4	20	21

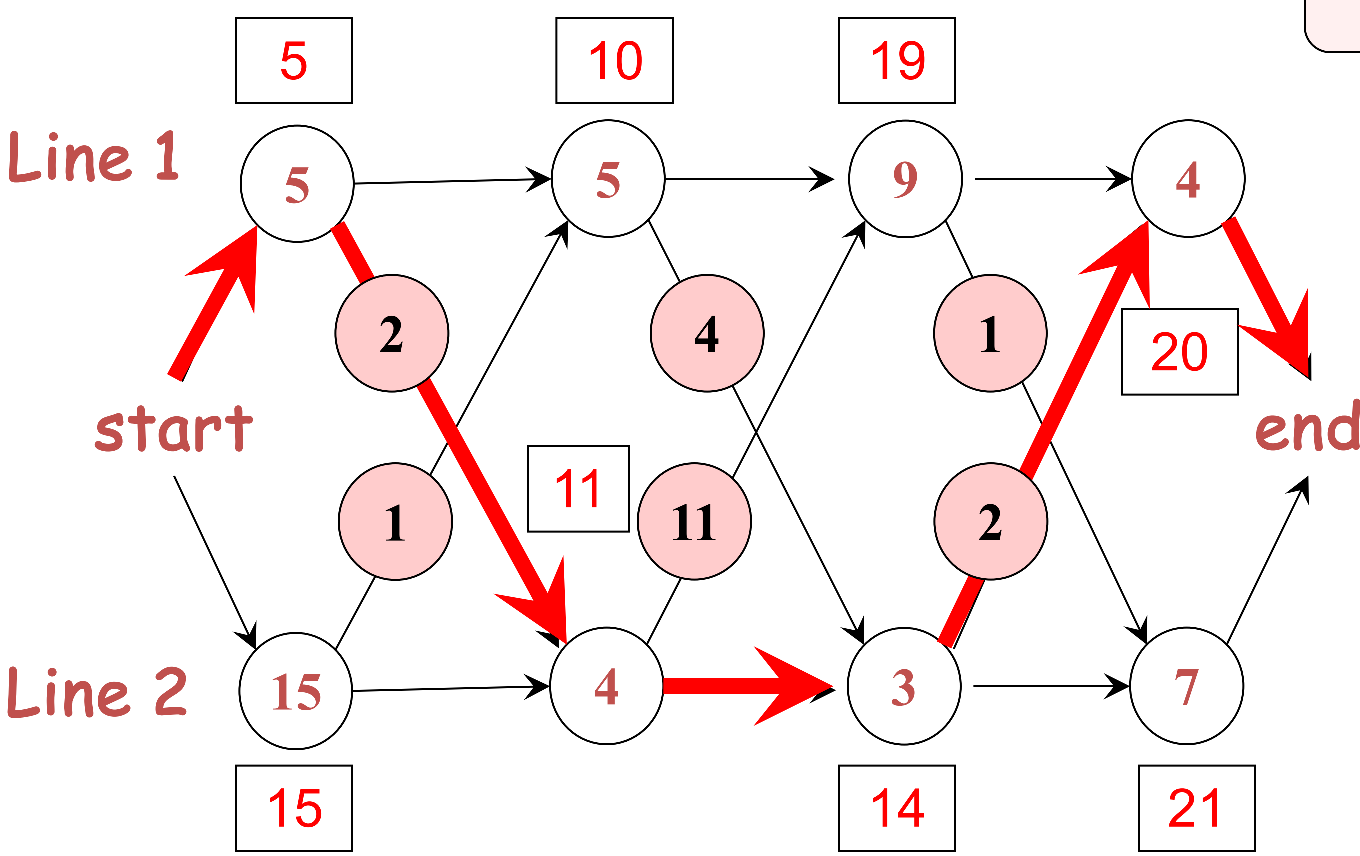
Summary

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

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

$$f^* = \min(\, f_1[n] \;,\; f_2[n] \,)$$

$f^* = 20$



j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3	19	14
4	20	21

Pseudo code

Time complexity is O(n)

set $f_1[1] = a_{1,1}$

set $f_2[1] = a_{2,1}$

for $j = 2$ to n **do**

begin

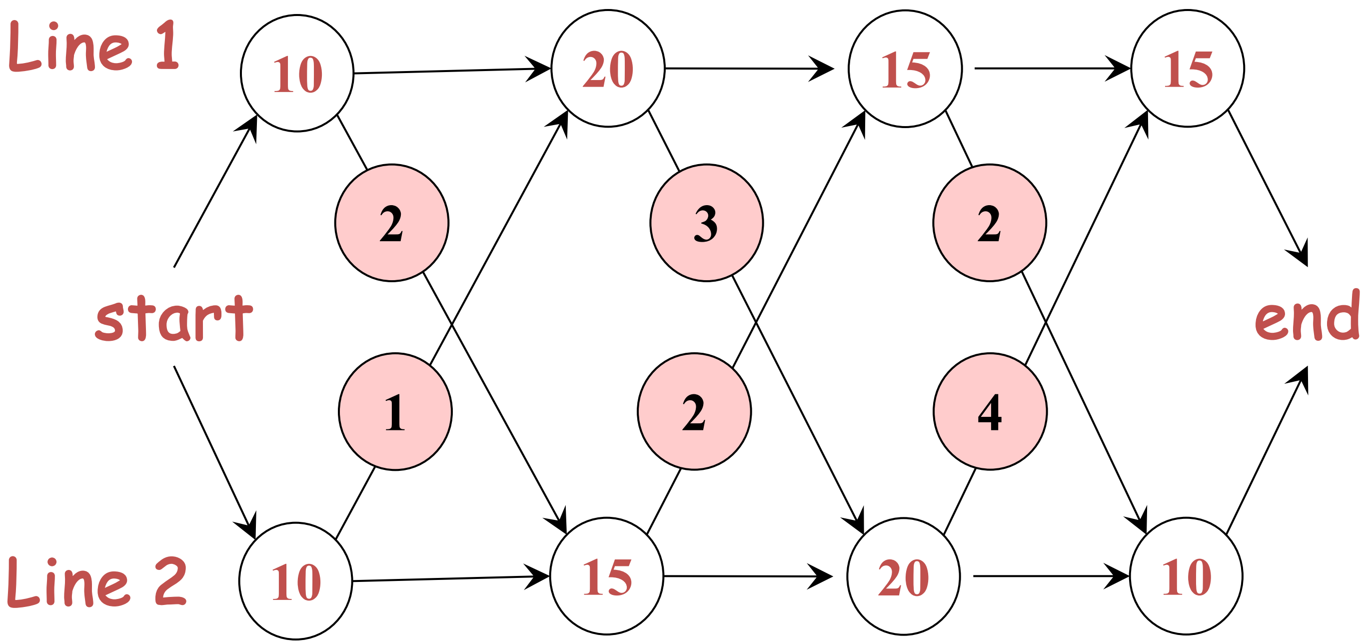
    set $f_1[j] = \min(\, \mathbf{f_1[j-1]+a_{1,j}} \;,\; \mathbf{f_2[j-1]+t_{2,j-1}+a_{1,j}} \,)$

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

end

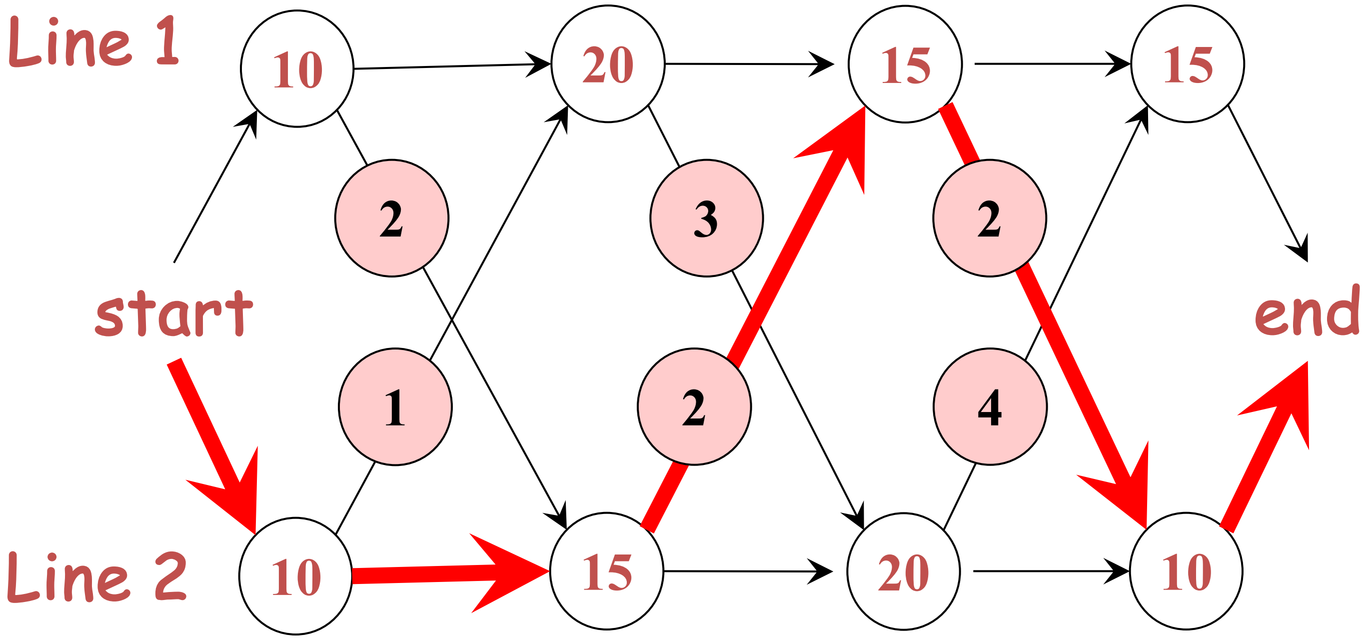
set $f^* = \min(f_1[\mathbf{n}]\,,\, f_2[\mathbf{n}])$

One more example



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

One more example – solution



f* = 54

j	$f_1[j]$	$f_2[j]$
1	10	10
2	30	25
3	42	45
4	57	54

What if there are 3 or more lines?

In general, m assembly lines: use multi-dimensional arrays.

a[i][j] – represents assemble time of station **j** on line **i**

t[i][j][k] – represents transfer time from station **j** on line **i** to station **(j+1)** on line **k**

– t[i][j][i] =0

f[i][j] – represents the best so far way of going thro' station **j** on line **i**

$$f[i][j] = \min_{1 \leq k \leq m} \left(\mathbf{f[k][j\text{-}1]+t[k][j\text{-}1][i] + a[i][j]} \right)$$

Pseudo code – calculate f[i][j]

optional

```
for i = 1 to m do set f[i][1] = a[i][1]
for j = 2 to n do begin // station by station
    for i = 1 to m do begin // line by line to find f[i][j]
        min_cost = f[1][j-1] + t[1][j-1][i]
        min_line = 1
        for k = 2 to m do begin
            if (f[k][j-1]+t[k][j-1][i] < min_cost) then begin
                min_cost = f[k][j-1]+t[k][j-1][i]
                min_line = k
            end
        end
        f[i][j] = min_cost + a[i][j]
        from_line[i][j] = min_line
    end
end
```

transfer from line 1 to line i

transfer from line k to line i

assume that t[i][j][i] = 0

Pseudo code – find optimal cost

optional

```
min_line = 1
min = f[1][n]
for i = 2 to m do
begin
    if (f[i][n] < min) then begin
        min_line = i
        min = f[i][n]
    end
end
f* = min
output f*
```

Pseudo code – find optimal path

optional

```
output "Station n: Line " + min_line
for j = n downto 2 do
begin
    min_line = from_line[min_line][j]
    output "Station " + (j-1) + ": Line " + min_line
end
```

Time Complexity

optional

$O(m)$: for i = 1 to m do set f[i][1] = a[i][1]

$O(nm^2)$: for j = 2 to n do begin

for i = 1 to m do begin

for k = 2 to m do begin

$O(n)$: for i = 2 to m do

$O(n)$: for j = n downto 2 do

Overall time complexity: **$O(nm^2)$**

– $O(m)+O(nm^2)+O(m)+O(n)$

Learning outcomes

- ✓ Understand the basic idea of dynamic programming
- ✓ Able to apply dynamic programming to compute Fibonacci numbers
- ✓ Able to apply dynamic programming to solve the assembly line scheduling problem

Dynamic programming

an efficient way to implement some divide and conquer algorithms