COMP108
Algorithmic Foundations

Dynamic Programming

Prudence Wong

http://www.csc.liv.ac.uk/~pwong/teaching/comp108/201617

Dynamic programming

an efficient way to implement some divide and conquer algorithms

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

Fibonacci numbers ...

Problem with recursive method

Fibonacci number F(n)

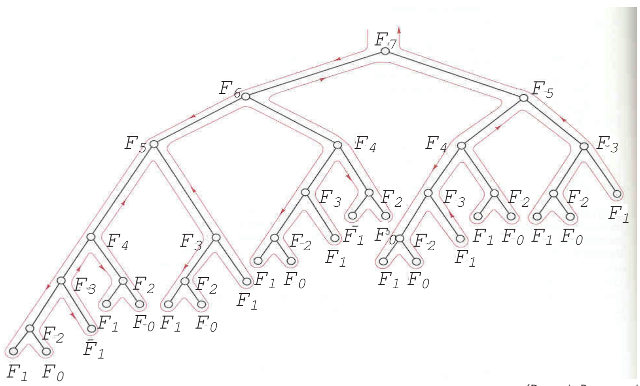
$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

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

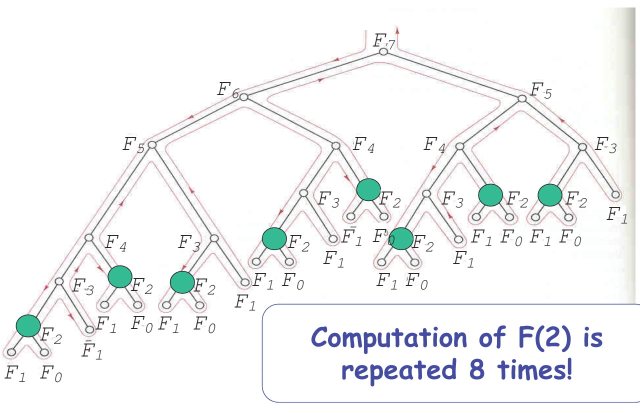
Pseudo code for the recursive algorithm:

```
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)

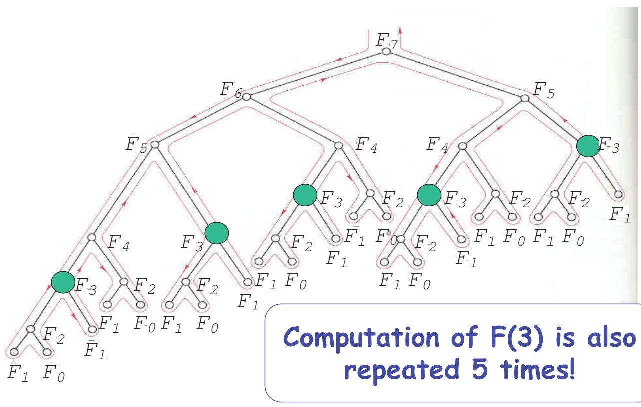


The execution of F(7)



Computation of F(2) is repeated 8 times!

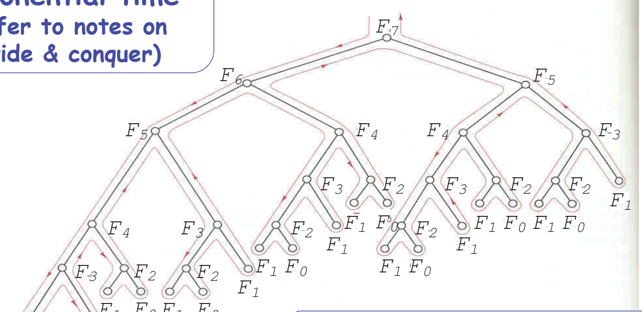
The execution of F(7)



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

The execution of F(7)

How long it takes?
exponential time
(refer to notes on divide & conquer)



Many computations are repeated!!

Idea for improvement

Memorization:

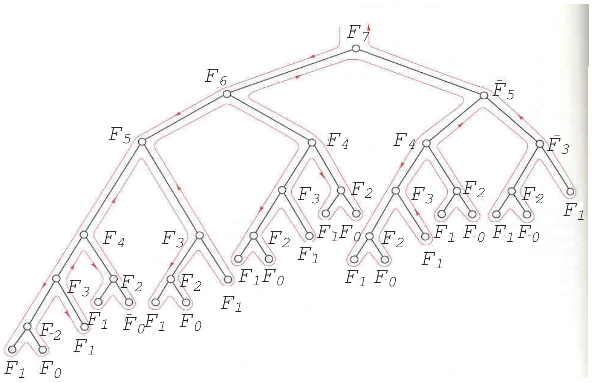
- Store F(i) somewhere after we have computed its value
- Afterward, we don't need to re-compute F(i); we can retrieve its value from our memory.

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

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

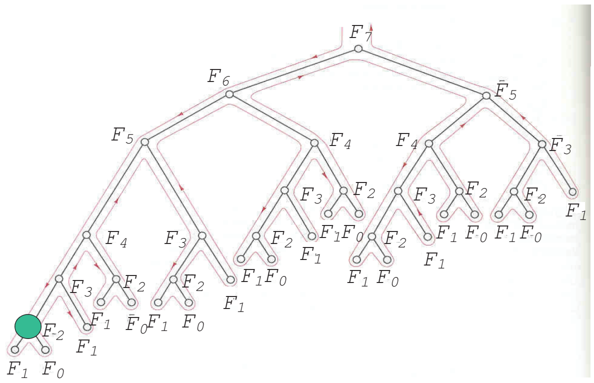
```
Main
  set v[0] = v[1] = 1
  for i = 2 to n do
    v[i] = -1
  output F(n)
```


Look at the execution of F(7)



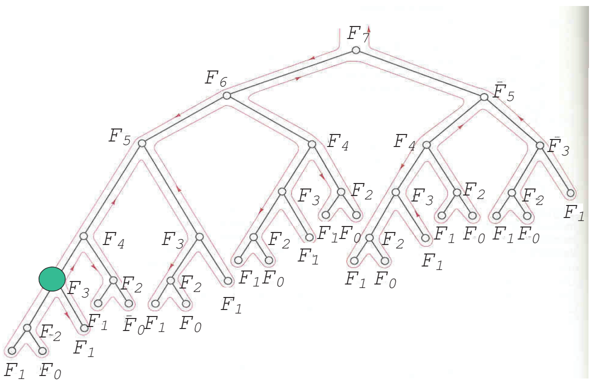
v[0]	1
v[1]	1
v[2]	-1
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Look at the execution of F(7)



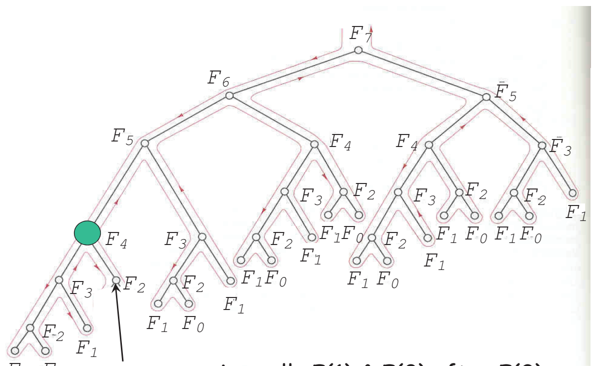
v[0]	1
v[1]	1
v[2]	2
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

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

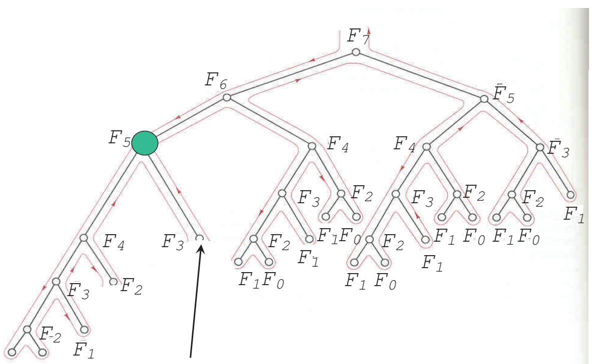
Look at the execution of F(7)



no recursive calls F(1) & F(0) after F(2)

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

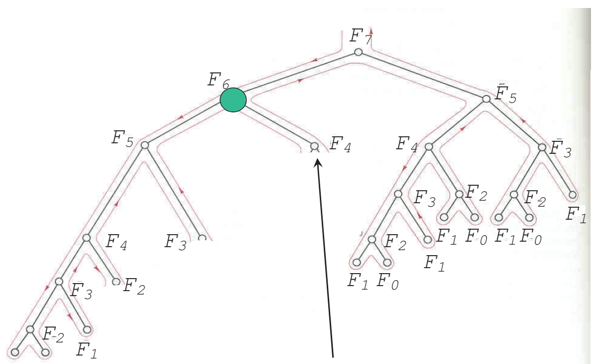
Look at the execution of F(7)



no recursive calls F(2) & F(1) after F(3)

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

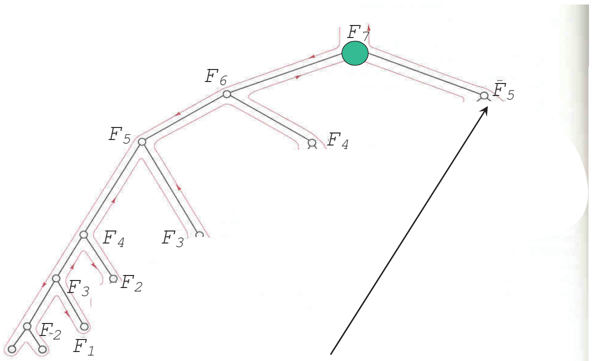
Look at the execution of F(7)



no recursive calls F(3) & F(2) after F(4)

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

no recursive calls F(4) & F(3) after F(5)

Can we do even better?

Observation

- The 2nd version still makes many function calls, and each wastes time 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. Write a recursive procedure to compute G(n)
2. Draw the execution tree of computing **G(6)** recursively
3. Using dynamic programming, write a pseudo code to compute G(n) efficiently
4. What is the time complexity of your algorithm?

Exercise

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

Recursive version:

```
Procedure G(n)
   if                        then
      return ??
   else return ??
```

Dynamic Programming version:

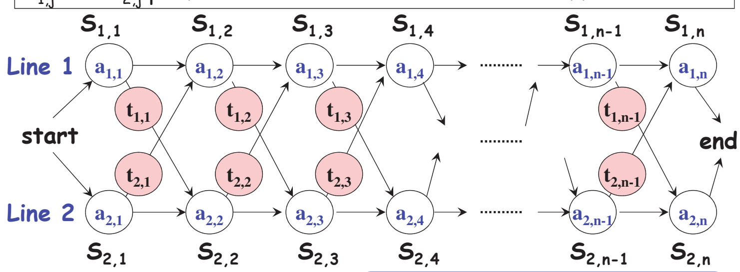
```
Procedure G(n)
```

O(??)

Assembly line scheduling ...

Assembly line scheduling

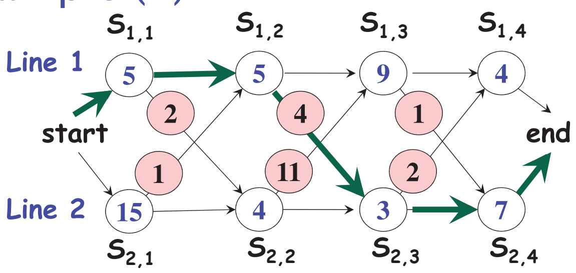
2 assembly lines, each with n stations ($S_{i,j}$: line i station j)
$S_{1,j}$ and $S_{2,j}$ perform same task but time taken is different



$a_{i,j}$: 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

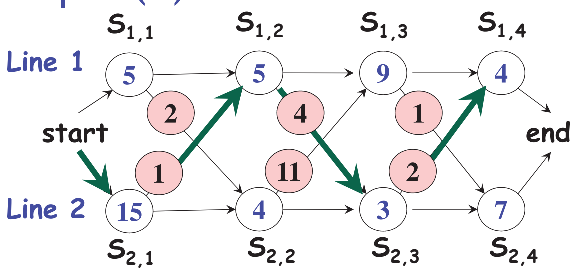
Example (1)



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

time required: 5 5 4 3 7 = 24

Example (2)



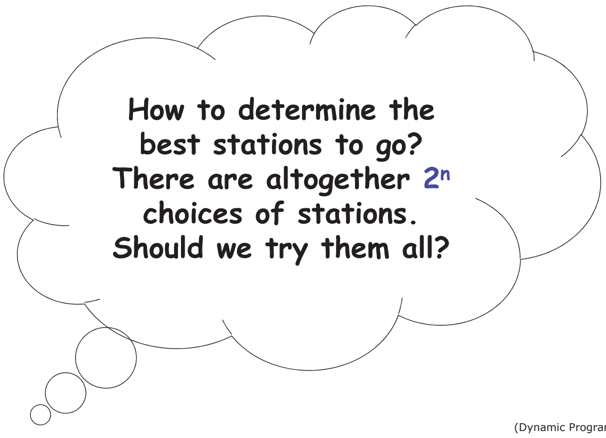
stations chosen: $S_{1,1}$ $S_{1,2}$ $S_{2,3}$ $S_{2,4}$

time required: 5 5 4 3 7 = 24

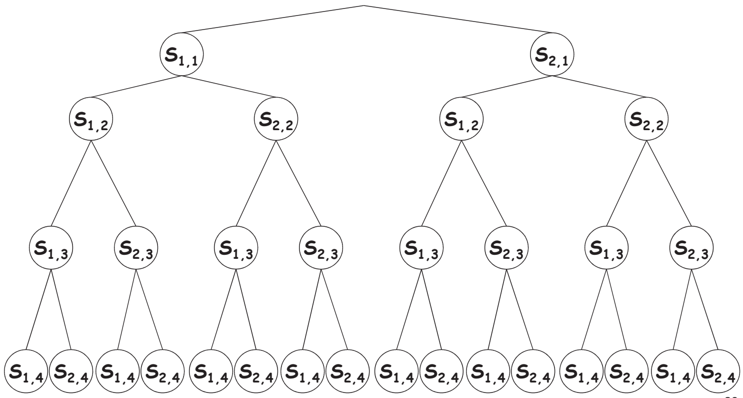
stations chosen: $S_{2,1}$ $S_{1,2}$ $S_{2,3}$ $S_{1,4}$

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

Example (2)

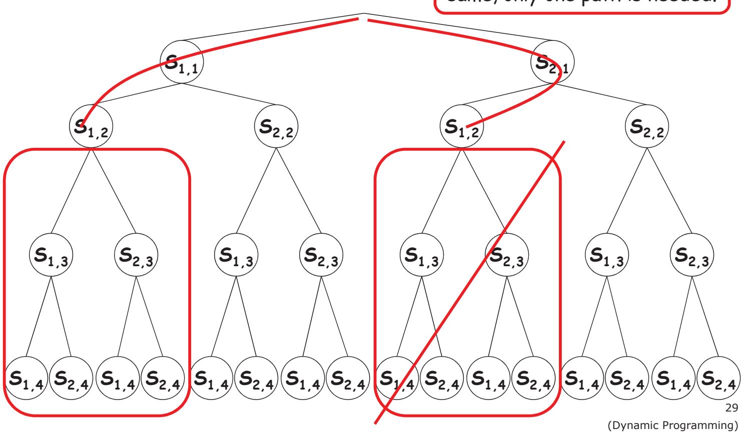


All possible choices



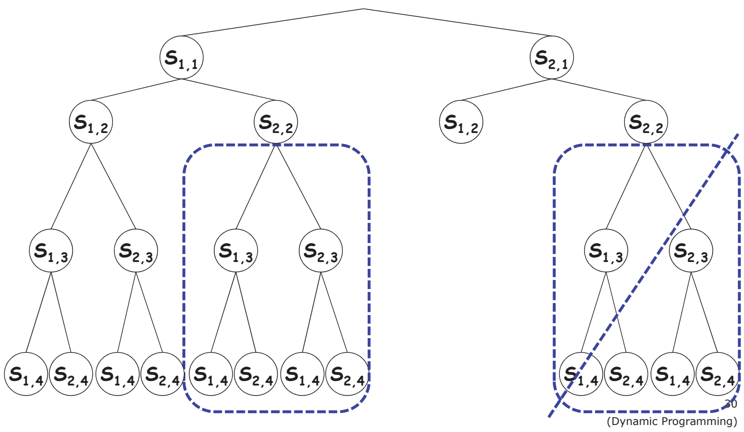
All possible choices

The two subtrees cost the same, only one path is needed.

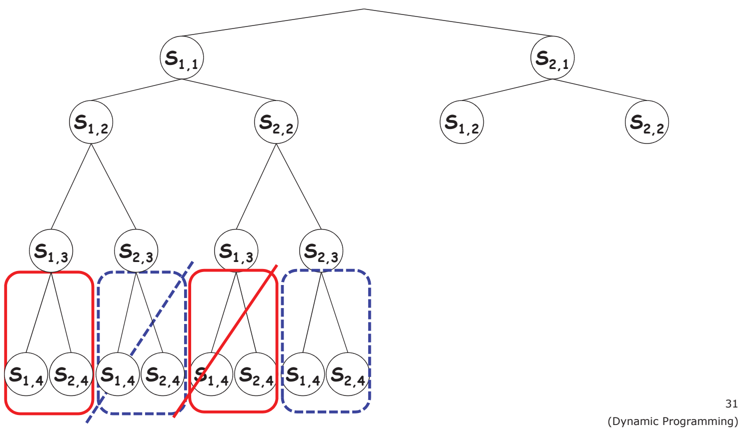


All possible choices

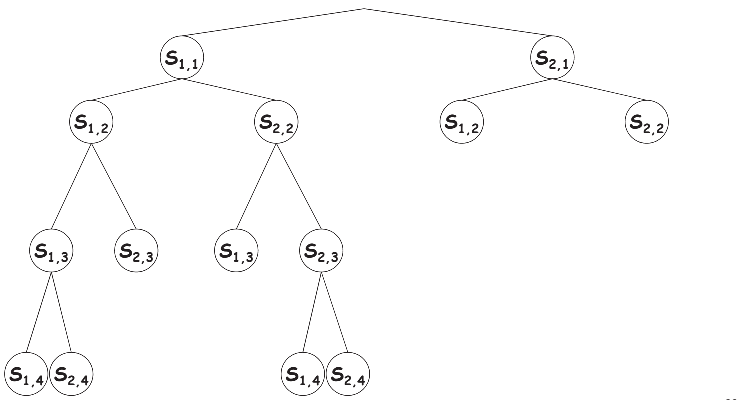
Similarly, ...



All possible choices



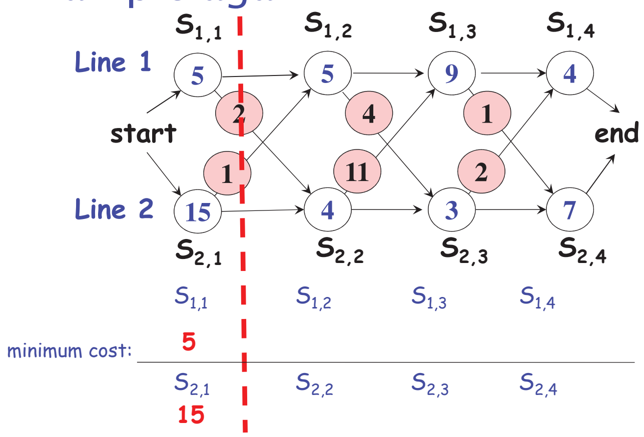
All possible choices



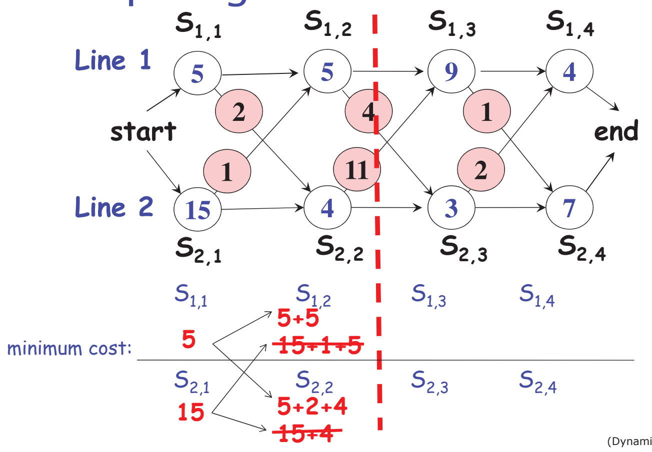
Good news: Dynamic Programming

- 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 $S_{1,n}$ and $S_{2,n}$, 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' $S_{1,n-1}$ and $S_{2,n-1}$
  3. In general, we want to know the fastest way to get thro' $S_{1,j}$ and $S_{2,j}$, for all j.

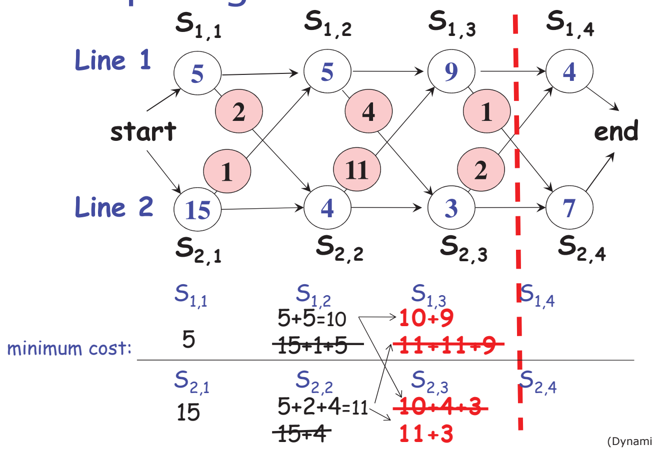
Example again



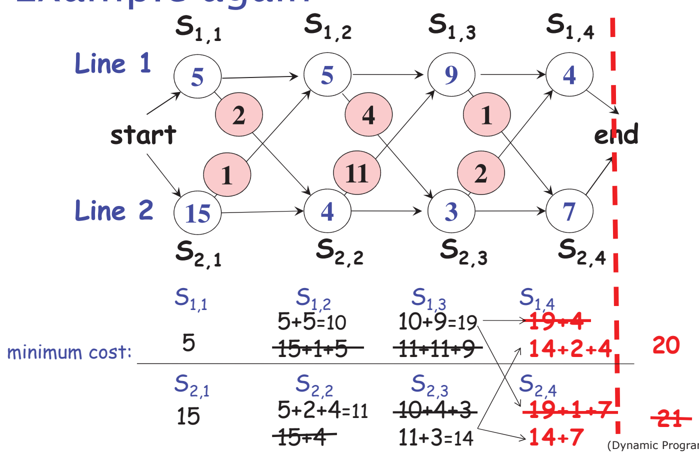
Example again



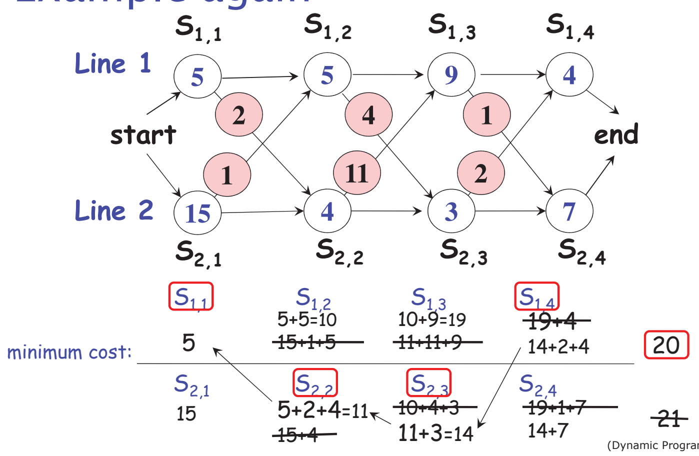
Example again



Example again



Example again



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:

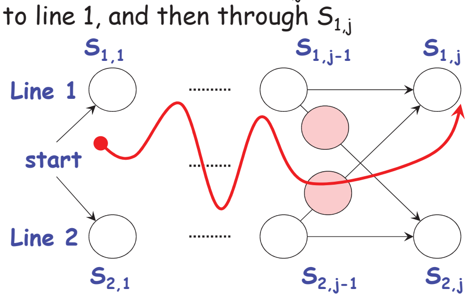
- Starting from $f_1[1]$ and $f_2[1]$, compute $f_1[j]$ and $f_2[j]$ incrementally

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 $S_{1,j}$

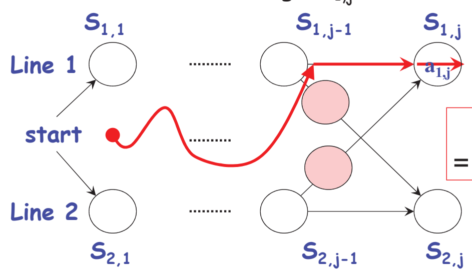


Solving the sub-problems (1)

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 $S_{1,j}$



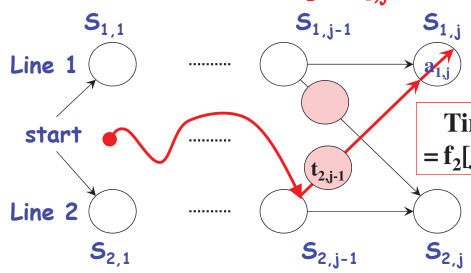
Time required $= f_1[j-1] + a_{1,j}$

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)

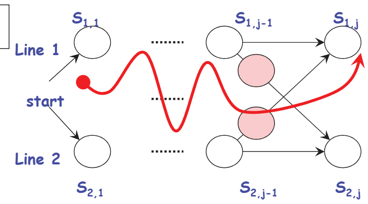
Q1: what is the fastest way to get thro' $S_{1,j}$?

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



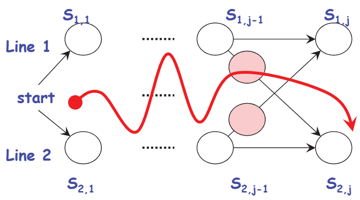
Solving the sub-problems (2)

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' $S_{2,j}$:

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

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

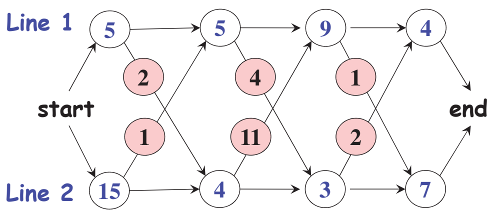


Summary

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

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

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



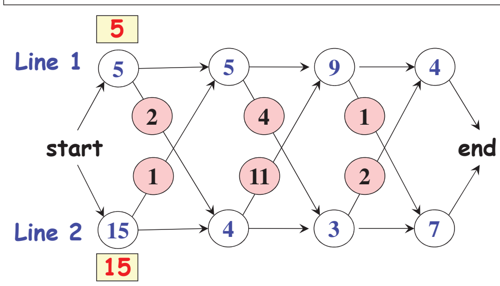
j	$f_1[j]$	$f_2[j]$
1		
2		
3		
4		

Summary

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

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

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



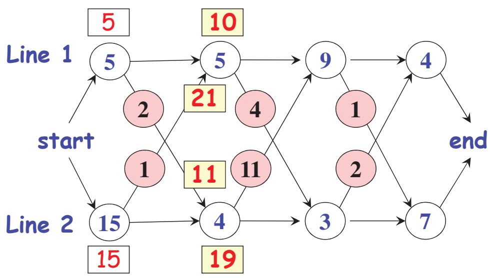
j	$f_1[j]$	$f_2[j]$
1	5	15
2		
3		
4		

Summary

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

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

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



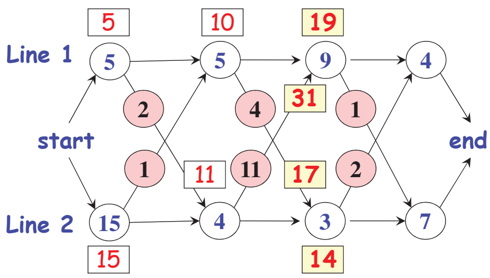
j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3		
4		

Summary

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

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

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



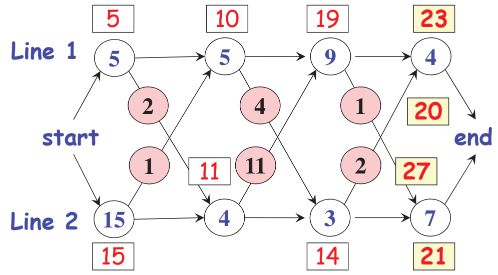
j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3	19	14
4		

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

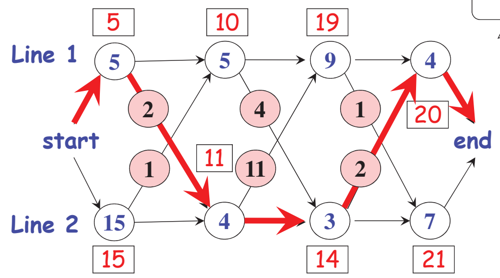
Summary

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

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

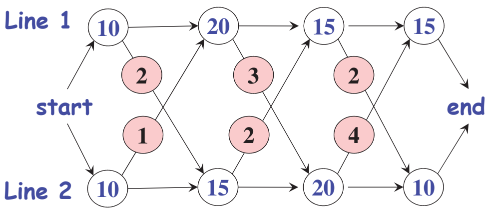
$$f^* = \min(f_1[n], f_2[n])$$

f* = 20



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

Exercise



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

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 $f_1[j] = \min(f_1[j-1]+a_{1,j},\ f_2[j-1]+t_{2,j-1}+a_{1,j})$
  set $f_2[j] = \min(f_2[j-1]+a_{2,j},\ f_1[j-1]+t_{1,j-1}+a_{2,j})$
end
set $f^* = \min(f_1[n], f_2[n])$

What about 3 assembly lines?