Dynamic Programming

Optimization Problems

- Some problems can have many possible/ feasible solutions with each solution having a specific cost. We wish to find the best solution with the optimal cost.
  - *Maximization problem* finds a solution with maximum cost
  - *Minimization problem* finds a solution with minimum cost
- A set of choices must be made in order to arrive at an optimal (min/max) solution, subject to some constraints.
- Is “Sorting a sequence of numbers” optimization problem?

Optimization Problems

- Two common techniques:
  - Greedy Algorithms (local)
    - Make the greedy choice and THEN
    - Solve sub-problem arising after the choice is made
    - The choice we make may depend on previous choices, but not on solutions to sub-problems
    - Top down solution, problems decrease in size
  - Dynamic Programming (global)
    - We make a choice at each step
    - The choice depends on solutions to sub-problems
    - Bottom up solution, smaller to larger sub-problems

Dynamic Programming

- Dynamic Programming was introduced by Richard Bellman in 1955.
- A strategy for designing algorithms
  - meta technique, not an algorithm
- Developed back in days when programming meant tabular method (like linear programming). Doesn't really refer to computer programming.
- Dynamic programming solves every sub problem just once.
  - Saves its answer in a table (array)
- More efficient than "*brute-force methods*" as it avoids the work of re-computing the answer every time the sub-problem is encountered.
- Dynamic programming is typically applied to **optimization problems**.

Dynamic Programming

- How to know if an optimization problem can be solved by applying dynamic programming?
- Two key ingredients
  - Optimal substructure
  - Overlapping sub-problems

- **Optimal Substructure:** A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within its optimal solutions to the sub-problems

- **Overlapping sub problems:** When a recursive algorithm revisits the same problem over and over again, then the optimization problem has ***overlapping sub-problems***

Dynamic Programming

Following steps are required in development of dynamic algorithms

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 in a bottom-up fashion
4. Construct an optimal solution from computed information

Note: Steps 1-3 form the basis of a dynamic programming solution to a problem. Step 4 can be omitted only if the value of an optimal solution is required.

Dynamic Programming

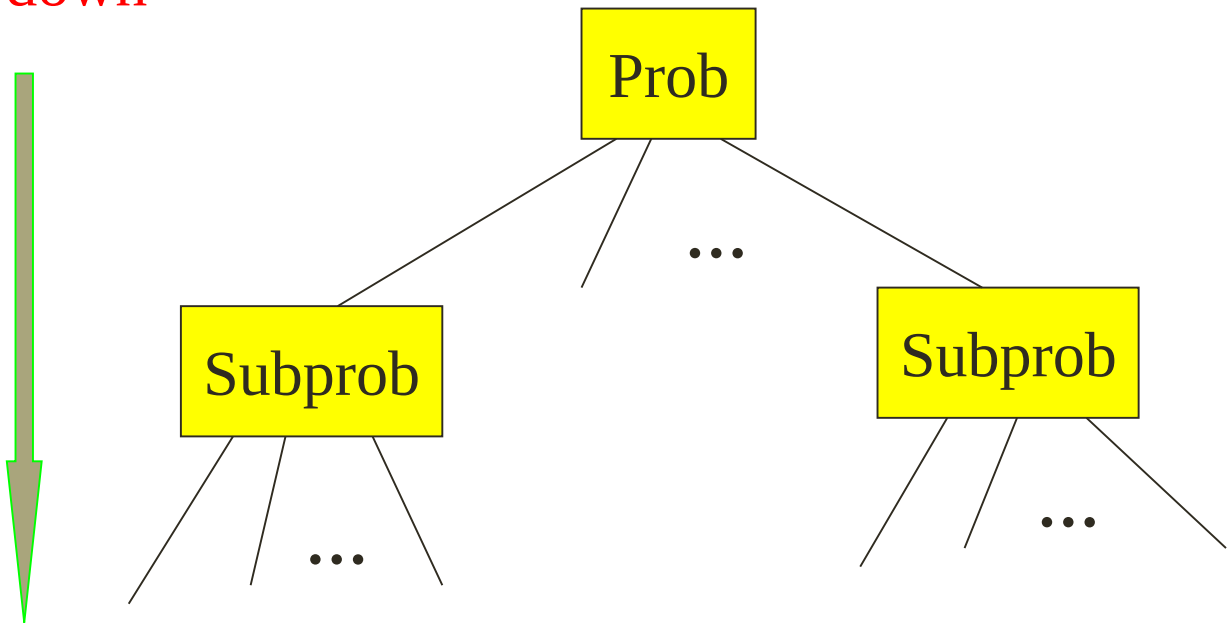
- Dynamic programming, like divide and conquer method, solves problems by combining the solutions to sub-problems.

- Divide and conquer algorithms:
  - partition the problem into independent sub-problem
  - Solve the sub-problem recursively and
  - Combine their solutions to solve the original problem

- In contrast, dynamic programming is applicable when the sub-problems are not independent.

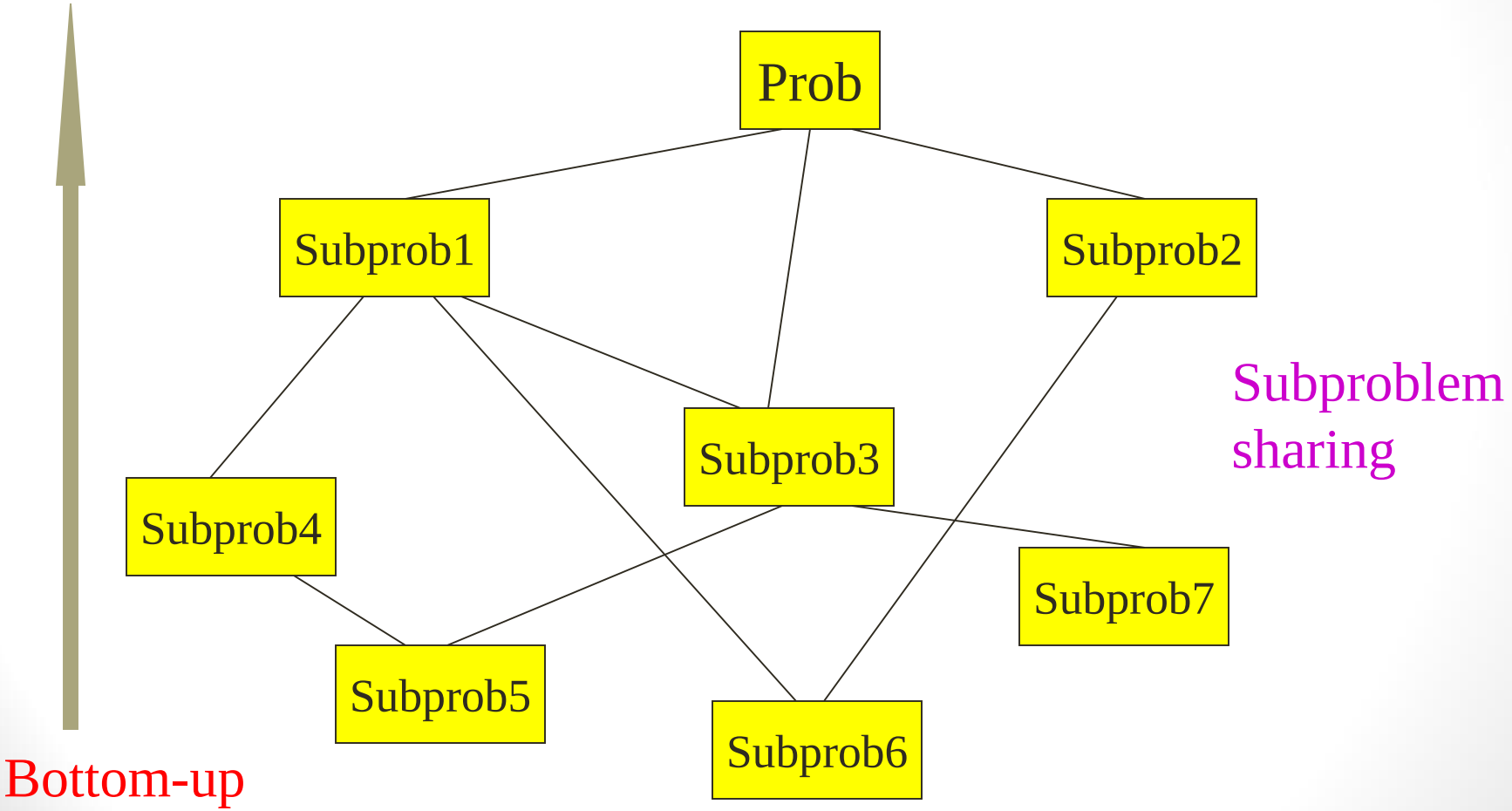
Divide-and-Conquer

Top-down



Independent sub-problems (no overlapping work).

Dynamic Programming



Knapsack Problem

Knapsack Problem

Given n items, each having a specific value v_i and weight w_i, and a knapsack of fixed capacity W, pack the knapsack with given items to maximize total value without exceeding the total capacity W of the knapsack.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack Problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

0-1 Knapsack Problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i and *W* are integer values)

- Problem: How to pack the knapsack to achieve maximum total value of packed items?

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

0-1 Knapsack Problem: Brute Force Approach

	Subset	*Total weight*	*Total value*
1.	*∅*	*0*	*0*
2.	{1}	2	20
3.	{2}	5	30
4.	{3}	10	50
5.	{4}	5	10
6.	{1,2}	7	50
7.	{1,3}	12	70
8.	{1,4}	7	30
9.	{2,3}	15	80
10.	{2,4}	10	40
11.	{3,4}	15	60
12.	{1,2,3}	17	not feasible
13.	{1,2,4}	12	60
14.	{1,3,4}	17	not feasible
15.	{2,3,4}	20	not feasible
16.	{1,2,3,4}	22	not feasible

#	*W*	*V*
1	2	20
2	5	30
3	10	50
4	5	10

knapsack capacity W = 16

Go through all combinations and find the one with maximum value and with total weight ≤ *W*

Running time = $O(2^n)$

0-1 Knapsack Problem: Brute Force Approach

Knapsack-BF (n, V, W, C)

Compute all subsets, s, of S = {1, 2, 3, 4}
forall s ∈ S
weight = Compute sum of weights of these items
if weight > C, not feasible
new solution = Compute sum of values of these items
solution = solution ∪ {new solution}
Return maximum of solution

Approach: In brute force algorithm, we go through all combinations and find the one with maximum value and with total weight less or equal to *W = 16*

Complexity

- Cost of computing subsets $O(2^n)$ for n elements
- Cost of computing weight = $O(2^n)$
- Cost of computing values = $O(2^n)$
- Total cost in worst case: $O(2^n)$

0-1 Knapsack problem: Dynamic Programming Approach

We can do better with an algorithm based on dynamic programming. We need to carefully identify/define the sub-problems

Let's try this:

- If items are labeled *1..n*, then a sub-problem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

This is a reasonable sub-problem definition. The question is can we describe the final solution (S_n) in terms of sub-problems (S_k)? Unfortunately, we can't do that.

Defining a Sub-problem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

?

Max weight: W = 20

For S_4:

Total weight: 14

Maximum benefit: 20

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4

S_5

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_5 = 9$ $b_5 = 10$

For S_5:

Total weight: 20

Maximum benefit: 26

Solution for S_4 is not part of the solution for S_5!!!

Defining a Sub-problem

As we have seen, the solution for S_4 is not part of the solution for S_5 so our definition of a sub-problem is flawed and we need another one!

Let's add another parameter: *w*, which will represent the maximum weight for each subset of items.

The sub-problem then will be to compute *V[k,w], i.e.,* to find an optimal solution for S_k = *{items labeled 1, 2, .. k} in a knapsack of size w*

Assuming knowing V[i, j], where i = 0, 1, 2, … k-1, and j = 0, 1, 2, …w, how to derive V[k,w]?

Recursive Formula for sub-problems

Recursive formula for sub-problems:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

The best subset of S_k that has the total weight $\leq w$, either contains item k or not.

- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

```
for w = 0 to W
   V[0,w] = 0
for i = 1 to n
   V[i,0] = 0
for i = 1 to n
   for w = 0 to W
         if w_i <= w // item i can be part of the solution
               if b_i + V[i-1,w-w_i] > V[i-1,w]
                     V[i,w] = b_i + V[i-1,w- w_i]
               else
                     V[i,w] = V[i-1,w]
         else V[i,w] = V[i-1,w]  // w_i > w
```

O(W)

O(n)

Repeat *n* times

O(W)

Remember that the brute-force algorithm takes $O(2^n)$

What is the running time of this algorithm? O(n*W)

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

V[0,w] = 0

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
for i = 1 to n
        V[i,0] = 0
```

Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if w_i <= w // item i can be part of the solution

    if $b_i + V[i-1,w-w_i] > V[i-1,w]$

        $V[i,w] = b_i + V[i-1,w- w_i]$

    else

        V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // $w_i > w$

Example (5)

Items:

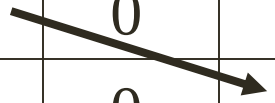
1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					



$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if w_i <= w // item i can be part of the solution

    if $b_i + V[i-1,w-w_i] > V[i-1,w]$

        $V[i,w] = b_i + V[i-1,w- w_i]$

    else

        $V[i,w] = V[i-1,w]$

else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if w_i <= w // item i can be part of the solution

    if $b_i + V[i-1,w-w_i] > V[i-1,w]$

        $V[i,w] = b_i + V[i-1,w- w_i]$

    else

        $V[i,w] = V[i-1,w]$

else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=4

w-w_i =2

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (8)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

if $w_i <= w$ // item i can be part of the solution

    if $b_i + V[i-1,w-w_i] > V[i-1,w]$

        $V[i,w] = b_i + V[i-1,w- w_i]$

    else

        $V[i,w] = V[i-1,w]$

else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (9)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

i=2

b_i=4

w_i=3

w=1

w-w_i =-2

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (10)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

i=2
b_i=4
w_i=3
w=2
w-w_i =-1

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

i=2

b_i=4

w_i=3

w=4

w-w_i =1

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (13)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

i=2
b_i=4
w_i=3
w=5
w-w_i =2

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

i=3

b_i=5

w_i=4

w= 1..3

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 4$

$w- w_i=0$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

i=4

b_i=6

w_i=5

w= 1..4

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (18)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Comments

This algorithm only finds the max possible value that can be carried in the knapsack

- i.e., the value in V[n,W]

To know the items that make this maximum value, an addition to this algorithm is necessary

How to find actual Knapsack Items

All of the information we need is in the table.

$V[n,W]$ is the maximal value of items that can be placed in the Knapsack.

Let i=n and k=W

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$ // Assume the i^{th} item is not in the knapsack

// Could it be in the optimally packed knapsack?

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k= 5

b_i=6

w_i=5

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (2)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4
k= 5
b_i=6
w_i=5
V[i,k] = 7
V[i–1,k] =7

i=n, k=W
while i,k > 0
    if *V[i,k]* ≠ *V[i–1,k]* then
        mark the *i*th item as in the knapsack
        i = i–1, *k = k-w_i*
    else
        i = i–1

Finding the Items (3)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3

k= 5

b_i=5

w_i=4

V[i,k] = 7

V[i–1,k] =7

i=n, k=W

while i,k > 0

if *V[i,k]* ≠ *V[i–1,k]* then

mark the i^{th} item as in the knapsack

$i = i–1$, $k = k\text{-}w_i$

else

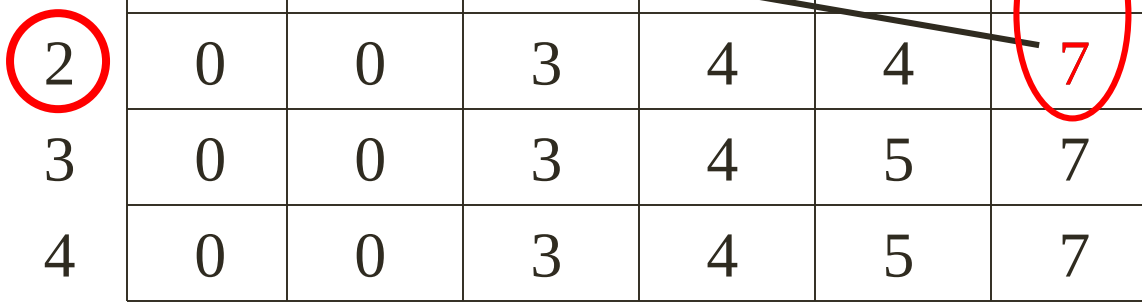
$i = i–1$

Finding the Items (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7



i=2

k= 5

b_i=4

w_i=3

V[i,k] = 7

V[i–1,k] =3

k – w_i=2

i=n, k=W

while i,k > 0

if *V[i,k]* ≠ *V[i–1,k]* then

mark the *i*th item as in the knapsack

i = i–1, *k = k-*w_i

else

i = i–1

Finding the Items (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=1

k= 2

b_i=3

w_i=2

$V[i,k] = 3$

$V[i-1,k] = 0$

$k - w_i = 0$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (6)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k= 0

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the n^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

The optimal knapsack should contain {1, 2}

Finding the Items (7)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

The optimal knapsack should contain {1, 2}

0-1 Knapsack Example

Input

- Given *n* items each
  - weight w_i
  - value v_i
- Knapsack of capacity *W*

Output: Find most valuable items that fit into the knapsack

Example:

item	*weight*	*value*	*knapsack capacity W* = 16
1	2	20	
2	5	30	
3	10	50	
4	5	10	

Exercise

1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

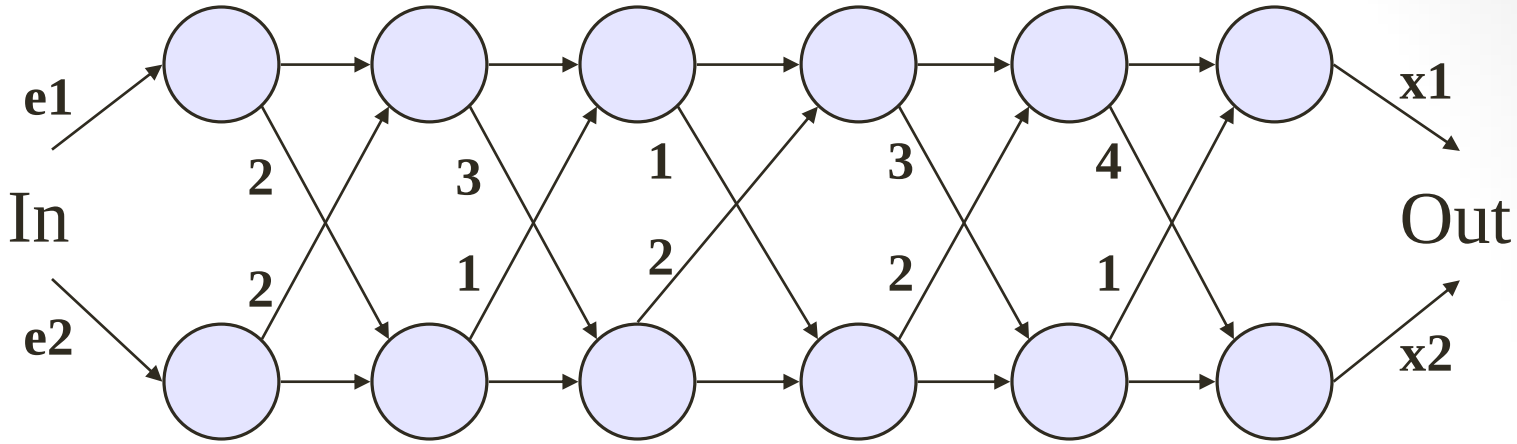
Assembly-Line Scheduling Problem

Assembly-Line Scheduling Problem

- Given two “parallel” assembly lines each with n stations.
- Each assembly line can perform any job
- An auto enters factory, goes through an assembly line, and exits
- The auto is served at n stations, each performing individual tasks
- Problem is to determine which stations to choose from lines 1 & 2 to minimize total time through the factory (i.e. the fastest route).

- After going through the jth station on a line i, the auto goes on to the (j+1)st station on either line.There is no transfer cost if it stays on the same line
- An optimal solution to the entire problem depends on optimal solutions to sub-problems
- We formulate a solution for the assembly line problem stage-by-stage

Notations: Assembly-Line Scheduling Problem



2 assembly lines, i = 1,2;

n stations, j = 1,...,n.

Stations $S_{i,j}$;

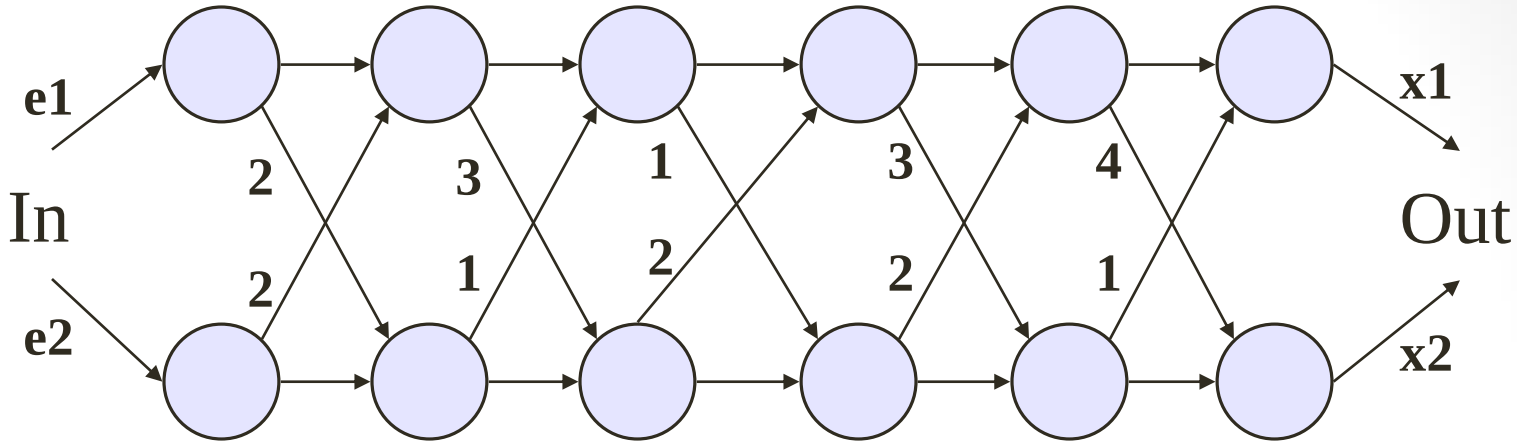
$a_{i,j}$ = assembly time at $S_{i,j}$;

$t_{i,j}$ = transfer time from $S_{i,j}$ (to $S_{i-1,j+1}$ OR $S_{i+1,j+1}$);

e_i = entry time from line i;

x_i = exit time from line i .

Brute Force Solution



Total Computational Time

= possible ways to enter in stations at level n x one way Cost

Possible ways to enter in stations at level 1 = 2^1

Possible ways to enter in stations at level 2 = $2^2 \cdots$

Possible ways to enter in stations at level n = 2^n

Total Computational Time = $\Omega(2^n)$

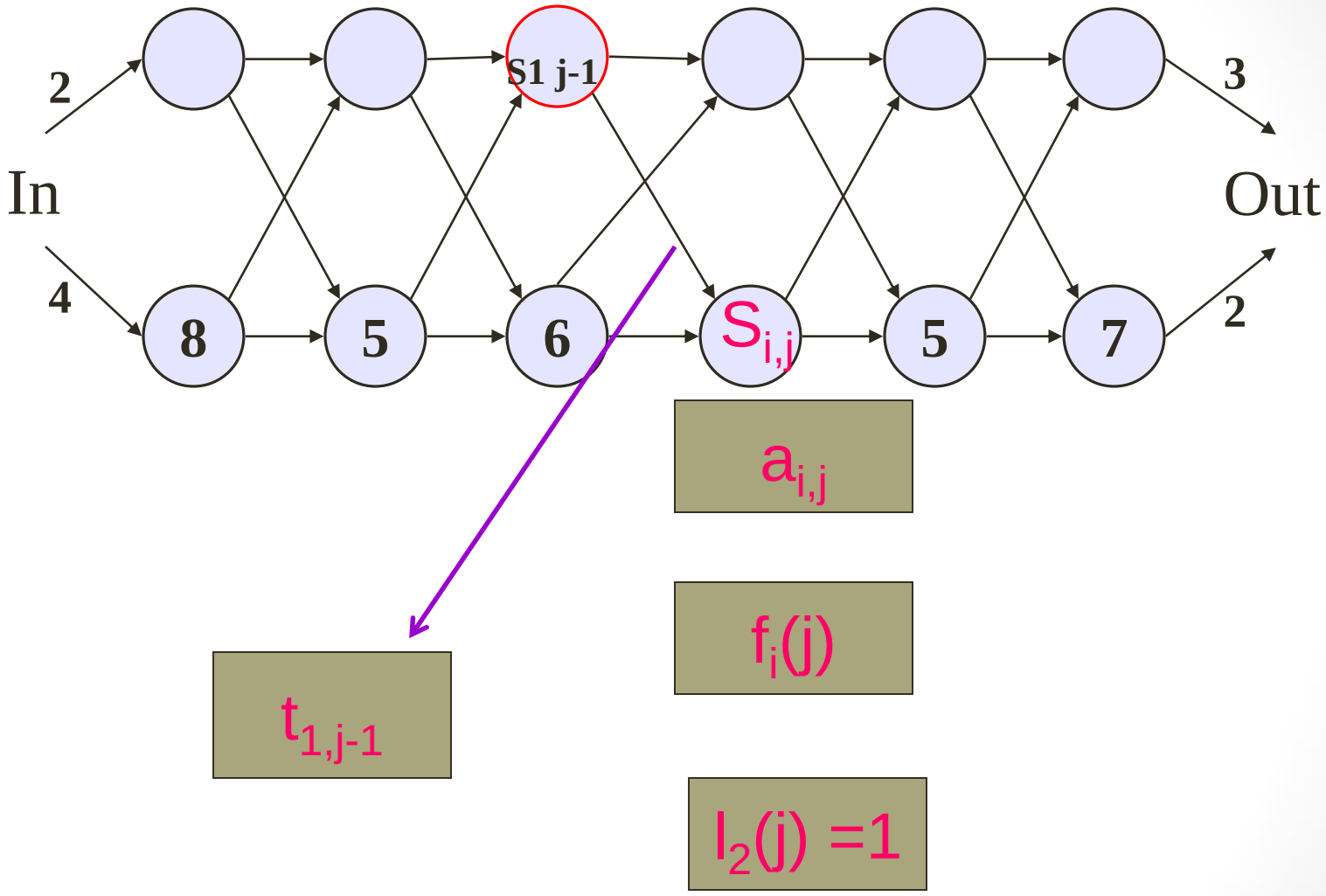
Dynamic Programming Solution

Notations: Finding Objective Functions

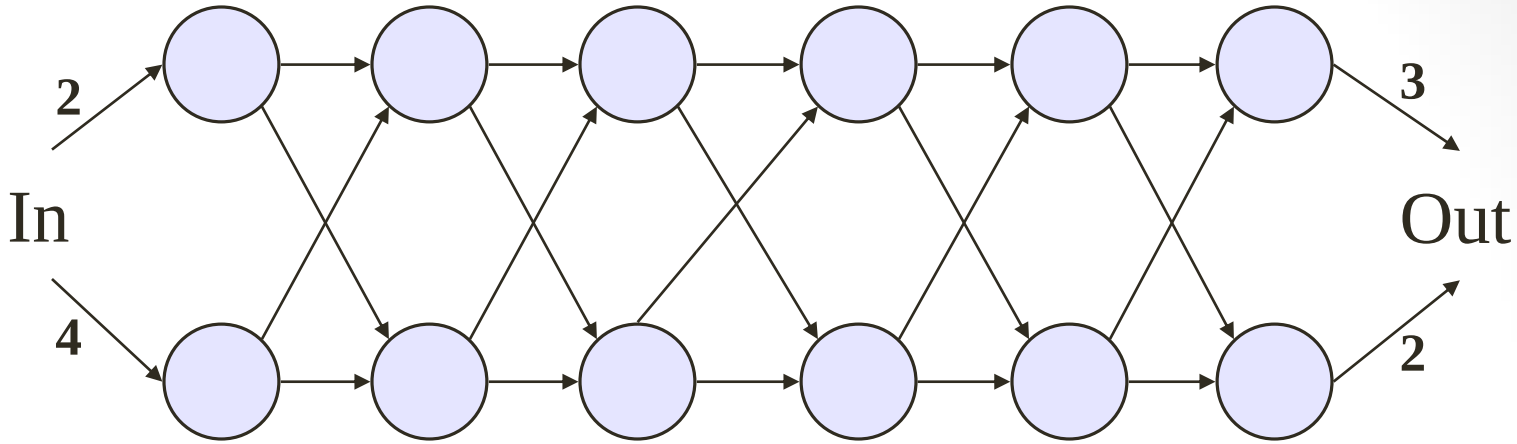
- Let $f_i[j]$ = fastest time from starting point to station $S_{i,j}$
- $f_1[n]$ = fastest time from starting point to station $S_{1\,n}$
- $f_2[n]$ = fastest time from starting point to station $S_{2\,n}$
- $l_i[j]$ = The line number, 1 or 2, whose station j-1 is used in a fastest way through station $S_{i,j}$.
- It is to be noted that $l_i[1]$ is not required to be defined because there is no station before 1
- $t_i[j-1]$ = transfer time from line i to station $S_{i-1,j}$ or $S_{i+1,j}$
- Objective function = $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$
- l^* = to be the line no. whose n^{th} station is used in a fastest way.

- Our improved strategy is to build a table to save previous results so that they don't have to be recomputed each time

Notations: Finding Objective Function



Mathematical Model: Finding Objective Function



$f_1[1] = e_1 + a_{1,1}$;

$f_2[1] = e_2 + a_{2,1}$.

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

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

Where $f_i[j]$ is the fastest possible time to $S_{i,j}$

$t_{i,j}$ is the transfer time from $S_{i,j}$

$a_{i,j}$ is the time at $S_{i,j}$

e_i is the entry time for assembly line i

Complete Model: Finding Objective Function

Base Cases

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

Two possible ways of computing $f_1[j]$

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

  For j = 2, 3, . . ., n

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

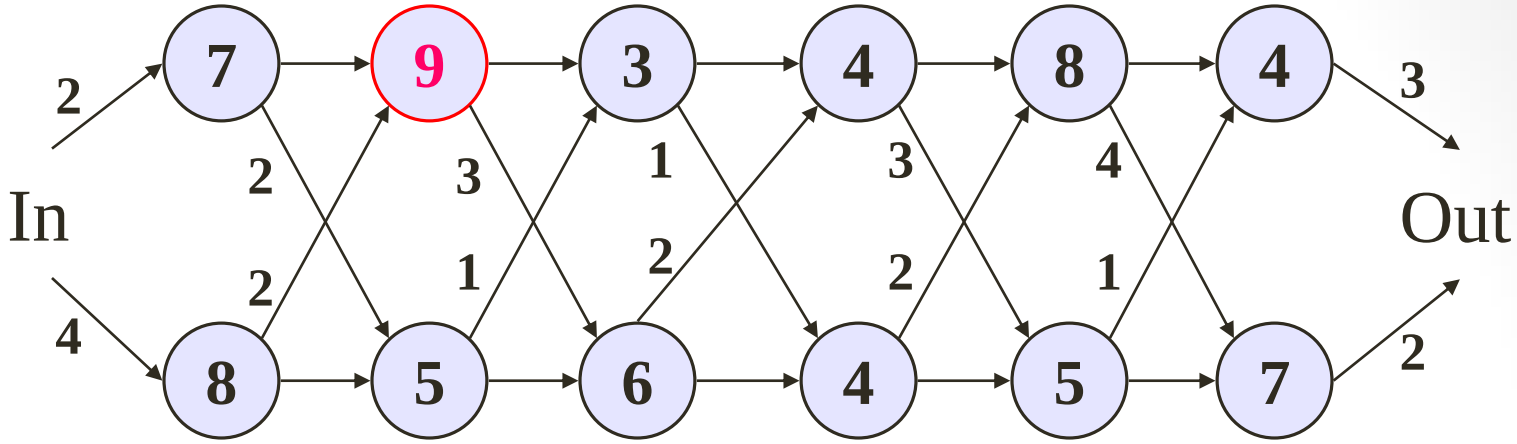
Symmetrically

  For j = 2, 3, . . ., n

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

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

Example: Computation of f1[2]



- $f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$
- $f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$

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

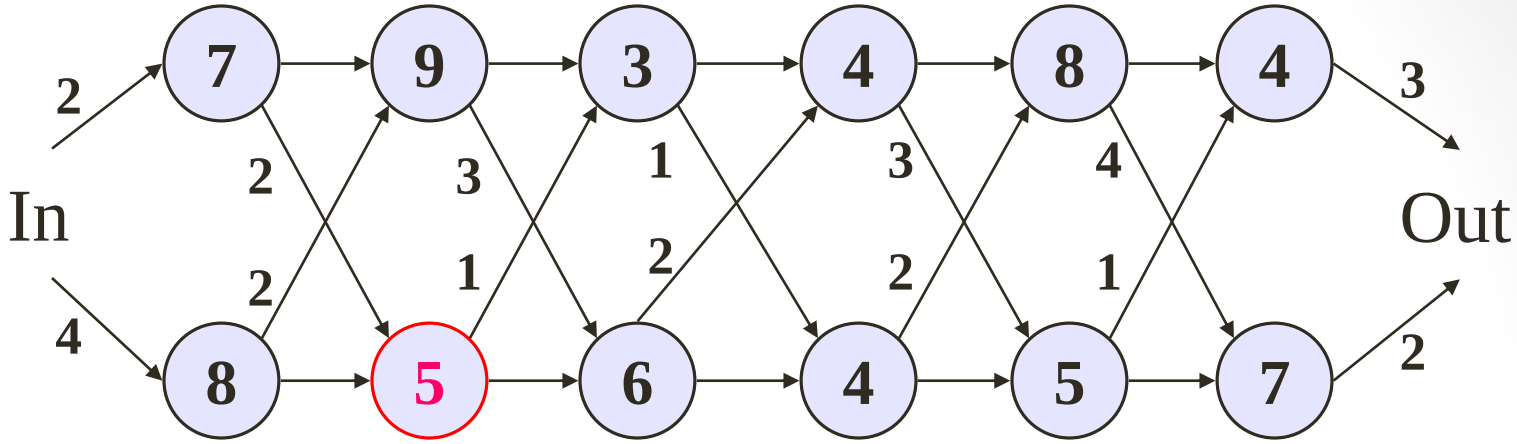
$f_2[j] = \min (f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 2$

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

$= \min (9 + 9, 12 + 2 + 9) = \min (18, 23) = 18, l_1[2] = 1$

Computation of f2[2]



- $f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$
- $f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$

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

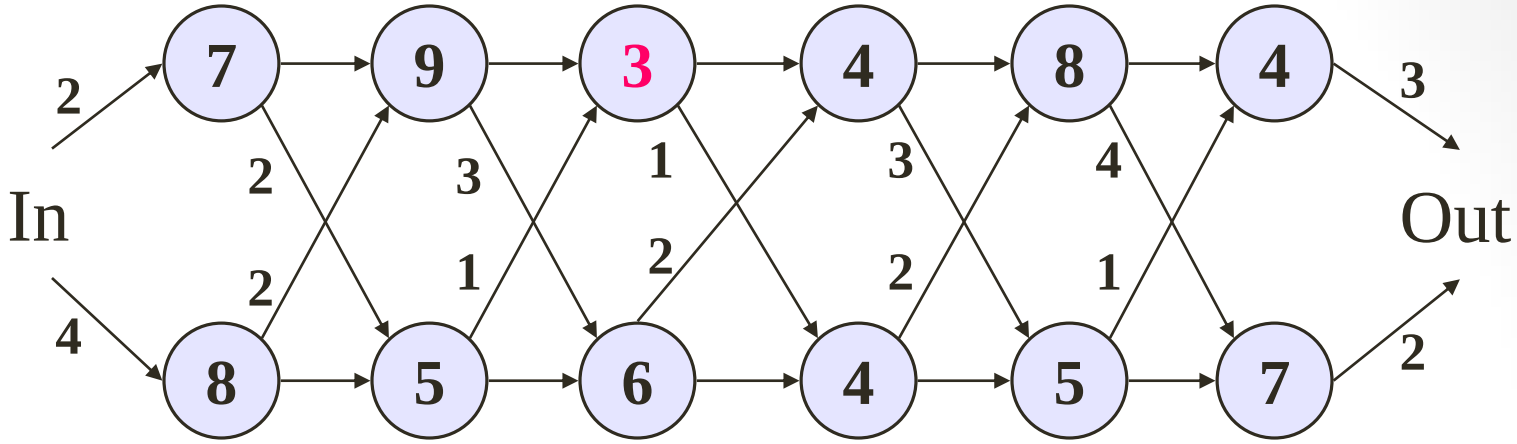
$f_2[j] = \min (f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 2$

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

$= \min (12 + 5, 9 + 2 + 5) = \min (17, 16) = 16, \quad l_2[2] = 1$

Computation of f1[3]



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

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

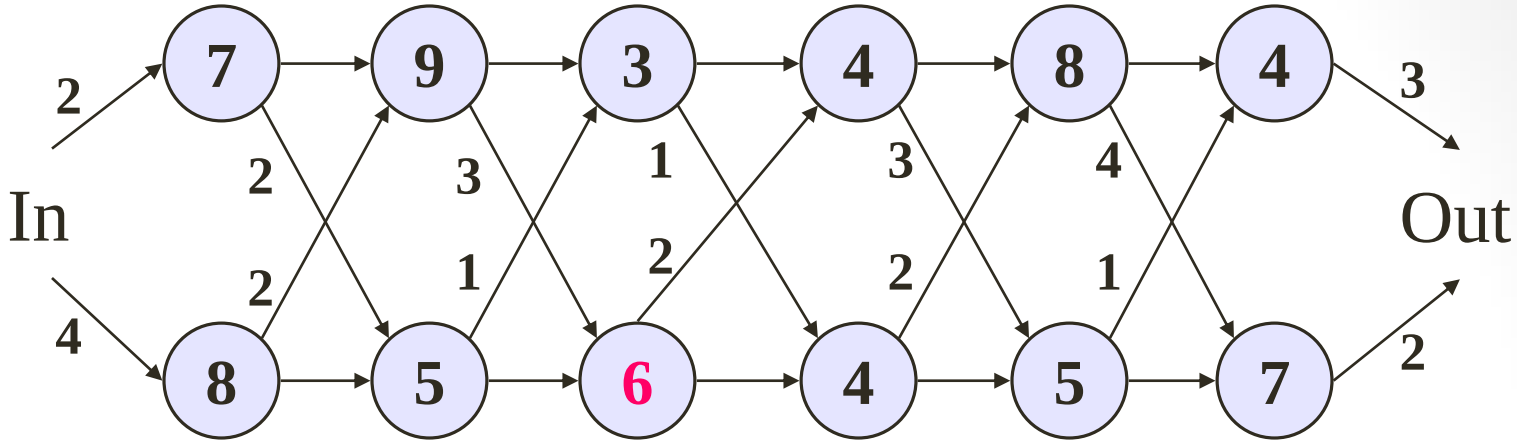
$j = 3$

$f_1[3] = \min (f_1[2] + a_{1,3}, f_2[2] + t_{2,2} + a_{1,3})$

$= \min (18 + 3, 16 + 1 + 3)$

$= \min (21, 20) = 20,$ $l_1[3] = 2$

Computation of f2[3]



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

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

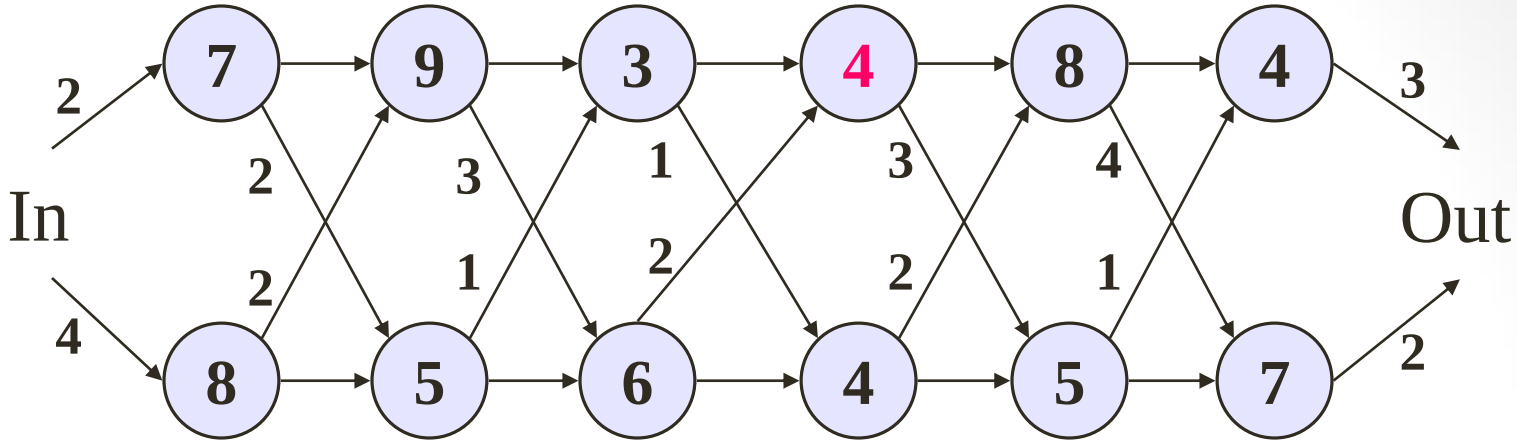
$j = 3$

$f_2[3] = \min (f_2[2] + a_{2,3}, f_1[2] + t_{1,2} + a_{2,3})$

$= \min (16 + 6, 18 + 3 + 6)$

$= \min (22, 27) = 22,$ $l_2[3] = 2$

Computation of f1[4]



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

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

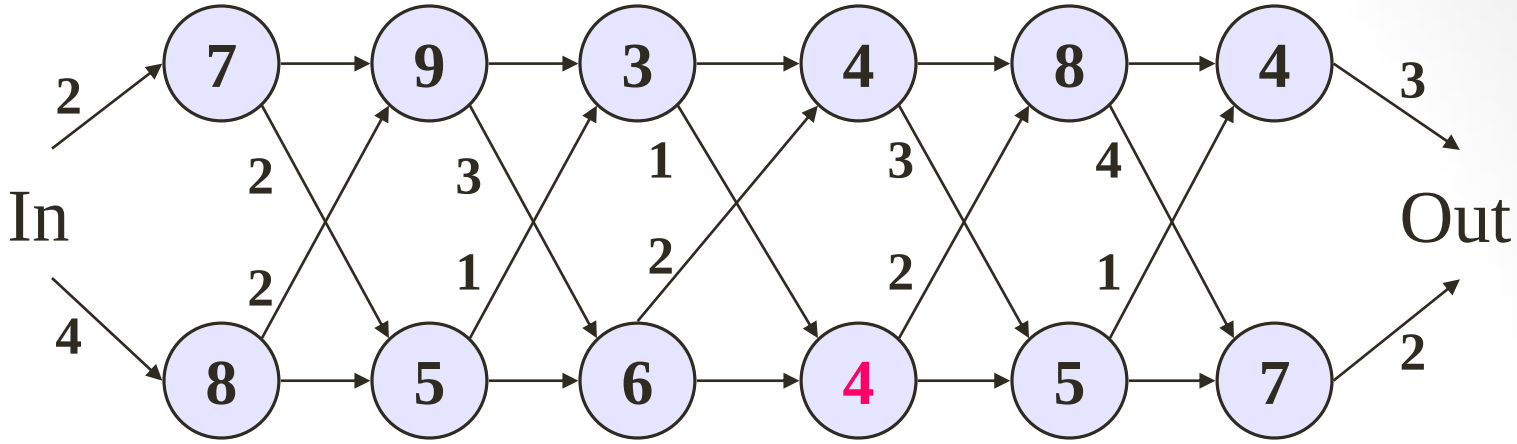
$j = 4$

$f_1[4] = \min (f_1[3] + a_{1,4}, f_2[3] + t_{2,3} + a_{1,4})$

$= \min (20 + 4, 22 + 1 + 4)$

$= \min (24, 27) = 24,$ $l_1[4] = 1$

Computation of f2[4]



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

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

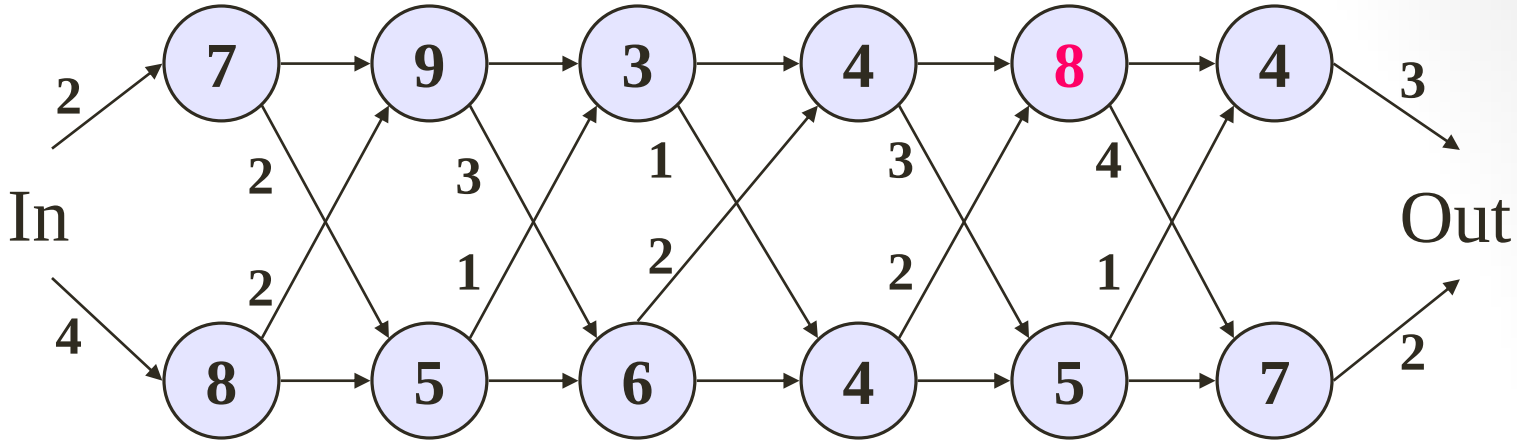
$j = 4$

$f_2[4] = \min (f_2[3] + a_{2,4}, f_1[3] + t_{1,3} + a_{2,4})$

$= \min (22 + 4, 20 + 1 + 4)$

$= \min (26, 25) = 25,$ $l_2[4] = 1$

Computation of f1[5]



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

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

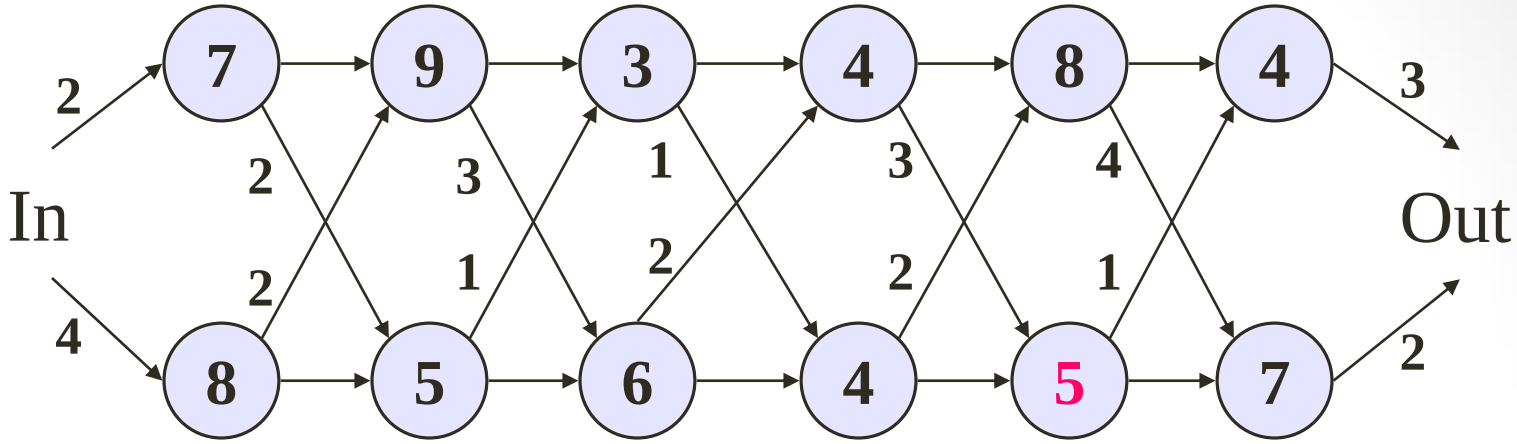
$j = 5$

$f_1[5] = \min (f_1[4] + a_{1,5}, f_2[4] + t_{2,4} + a_{1,5})$

$= \min (24 + 8, 25 + 2 + 8)$

$= \min (32, 35) = 32,$ $l_1[5] = 1$

Computation of f2[5]



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

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

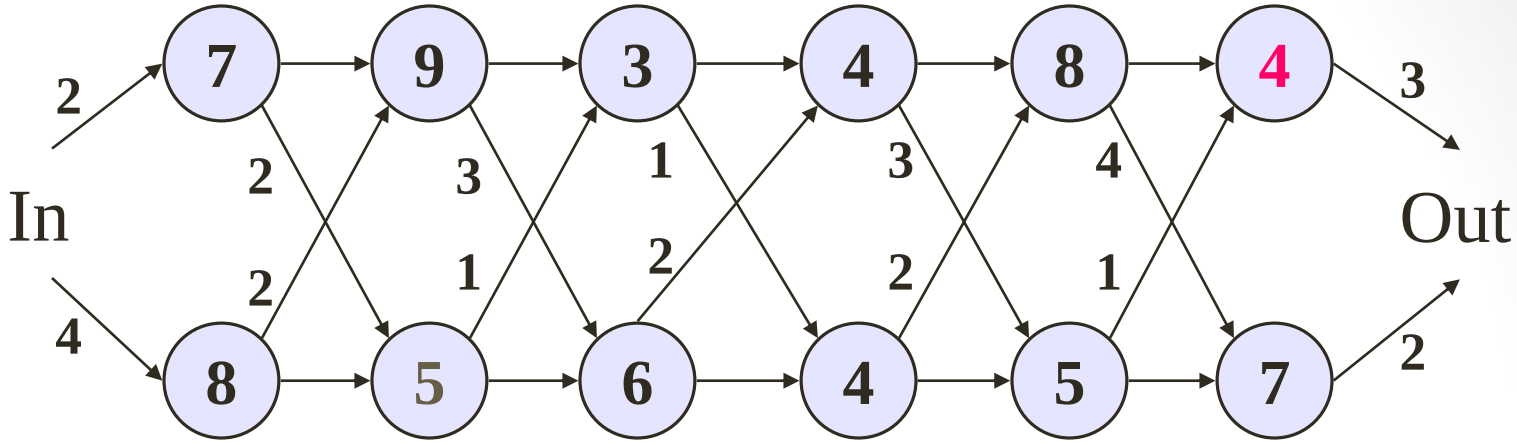
$j = 5$

$f_2[5] = \min (f_2[4] + a_{2,5}, f_1[4] + t_{1,4} + a_{2,5})$

$= \min (25 + 5, 24 + 3 + 5)$

$= \min (30, 32) = 30,$ $l_2[5] = 2$

Computation of f1[6]



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

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

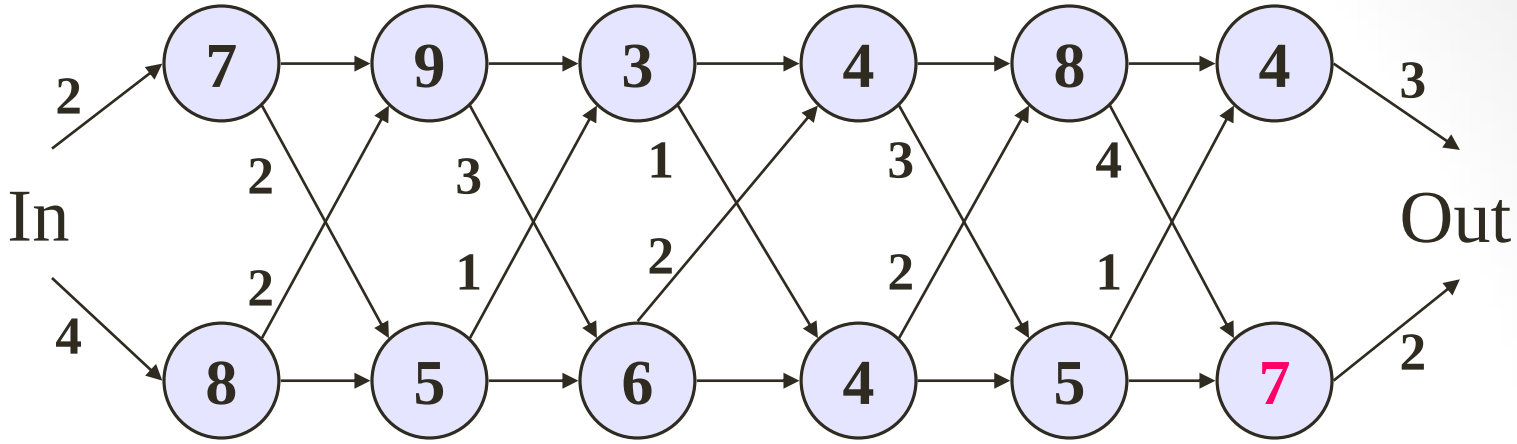
$j = 6$

$f_1[6] = \min (f_1[5] + a_{1,6}, f_2[5] + t_{2,5} + a_{1,6})$

$= \min (32 + 4, 30 + 1 + 4)$

$= \min (36, 35) = 35,$ $l_1[6] = 2$

Computation of f2[6]



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

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

$j = 6$

$f_2[6] = \min (f_2[5] + a_{2,6}, f_1[5] + t_{1,5} + a_{2,6})$

$= \min (30 + 7, 32 + 4 + 7)$

$= \min (37, 43) = 37,$ $l_2[6] = 2$

Keeping Track Constructing Optimal Solution

$f^* = \min (f_1[6] + x_1, f_2[6] + x_2)$

$= \min (35 + 3, 37 + 2)$

$= \min (38, 39) = 38$

$l^* = 1$

$l^* = 1 \Rightarrow$ Station $S_{1,6}$

$l_1[6] = 2 \Rightarrow$ Station $S_{2,5}$

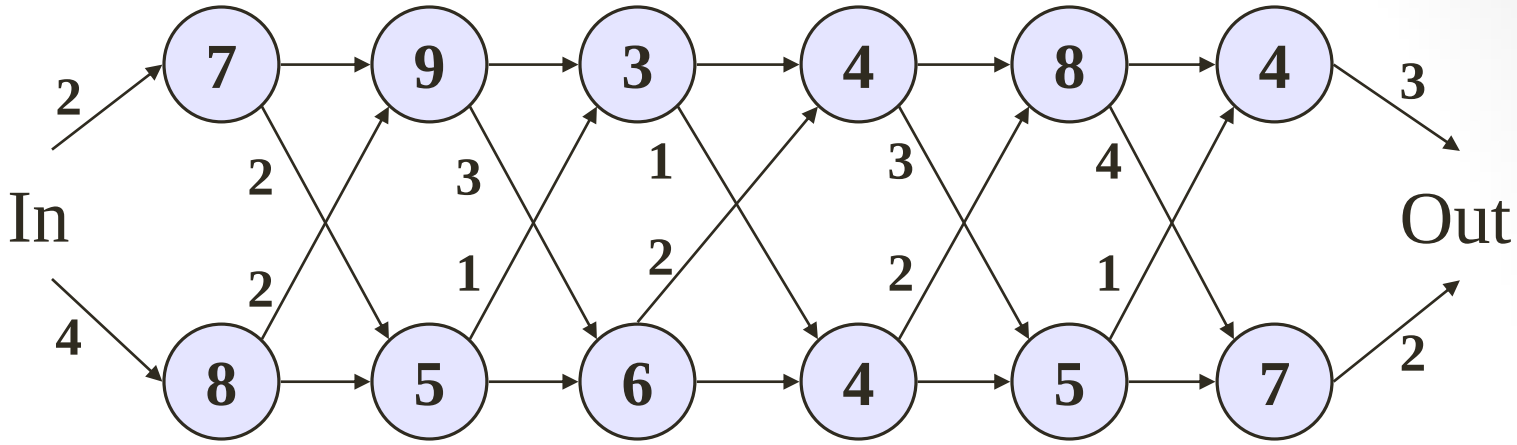
$l_2[5] = 2 \Rightarrow$ Station $S_{2,4}$

$l_2[4] = 1 \Rightarrow$ Station $S_{1,3}$

$l_1[3] = 2 \Rightarrow$ Station $S_{2,2}$

$l_2[2] = 1 \Rightarrow$ Station $S_{1,1}$

Entire Solution Set: Assembly-Line Scheduling



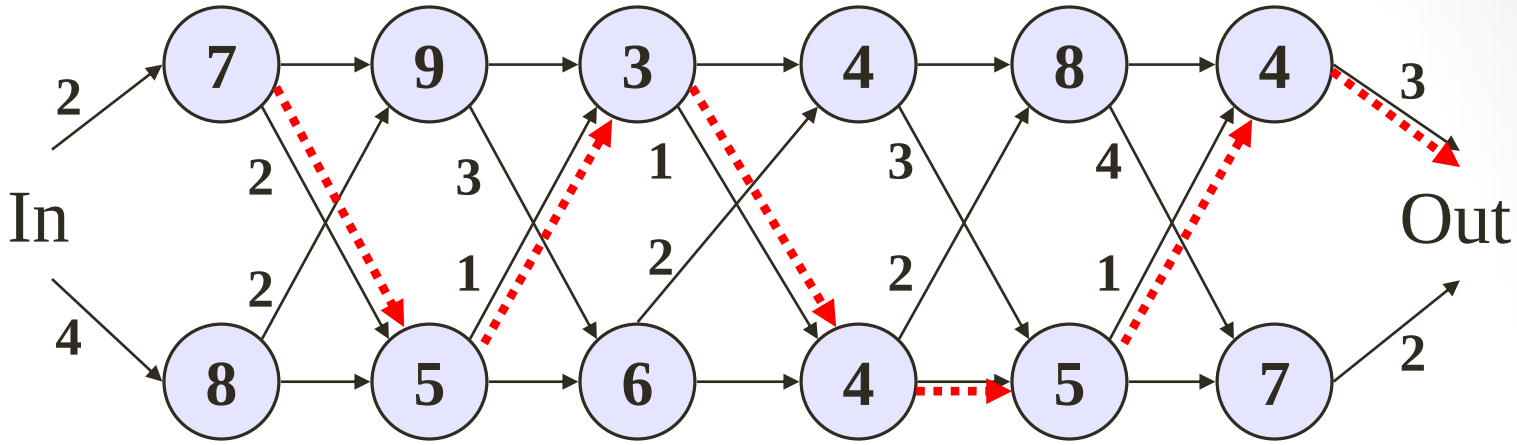
$f_i(j)$ \ j	1	2	3	4	5	6
1	9	18	20	24	32	35
2	12	16	22	25	30	37

f* = 38

$l_i(j)$ \ j	2	3	4	5	6
1	1	2	1	1	2
2	1	2	1	2	2

l* = 1

Fastest Way: Assembly-Line Scheduling



$l^* = 1 \Rightarrow$ Station $S_{1,6}$

$l_1[6] = 2 \Rightarrow$ Station $S_{2,5}$

$l_2[5] = 2 \Rightarrow$ Station $S_{2,4}$

$l_2[4] = 1 \Rightarrow$ Station $S_{1,3}$

$l_1[3] = 2 \Rightarrow$ Station $S_{2,2}$

$l_2[2] = 1 \Rightarrow$ Station $S_{1,1}$

Dynamic Algorithm

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** $f_1[j-1] + a_{1,j} \le f_2[j-1] + t_{2,j-1} + a_{1,j}$
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** $f_2[j-1] + a_{2,j} \le f_1[j-1] + t_{1,j-1} + a_{2,j}$
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** $f_1[n] + x_1 \le f_2[n] + x_2$
15. **then** $f^* = f_1[n] + x_1$
16. $l^* = 1$
17. **else** $f^* = f_2[n] + x_2$
18. $l^* = 2$

Total Computational Time = Θ(n)

Optimal Solution: Constructing The Fastest Way

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

This printout is "in reverse"; How can the print routine be changed to get a printout from Station 1 to Station 6 ?