DATA STRUCTURES AND ALGORITHMS

Dynamic Programming

Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

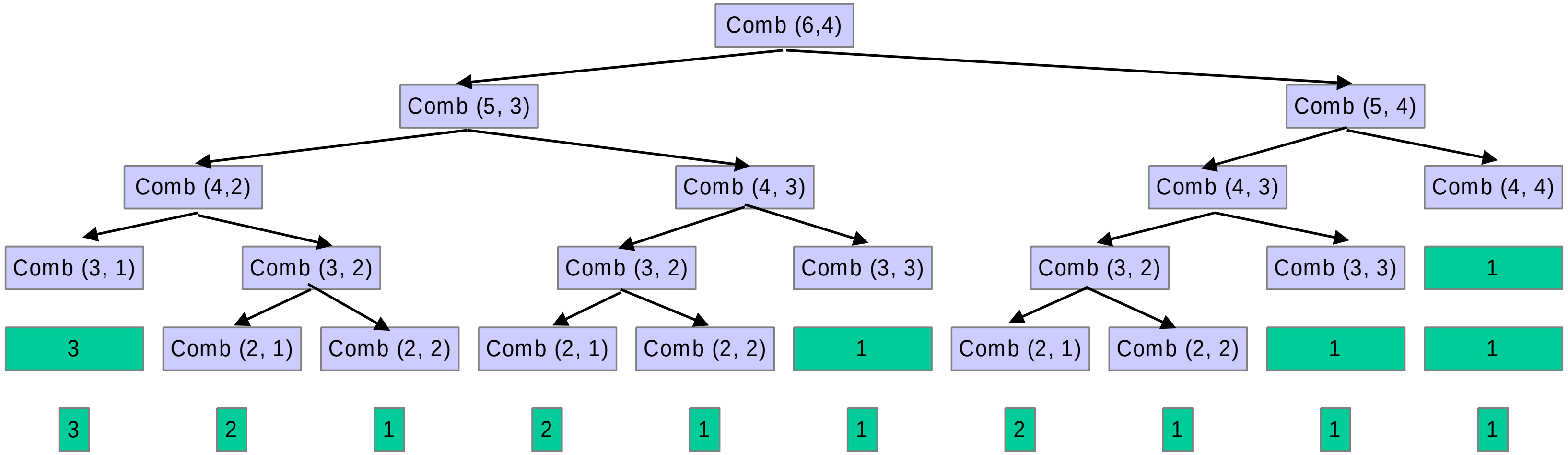
- Applicable when subproblems are not independent
  - Subproblems share subsubproblems
- E.g.: Combinations:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n}{1} = 1 \qquad \binom{n}{n} = 1$$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Example: Combinations



Comb (6,4)

Comb (5, 3) Comb (5, 4)

Comb (4,2) Comb (4, 3) Comb (4, 3) Comb (4, 4)

Comb (3, 1) Comb (3, 2) Comb (3, 2) Comb (3, 3) Comb (3, 2) Comb (3, 3) 1

3 Comb (2, 1) Comb (2, 2) Comb (2, 1) Comb (2, 2) 1 Comb (2, 1) Comb (2, 2) 1 1

3 2 1 2 1 1 2 1 1 1

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Dynamic Programming

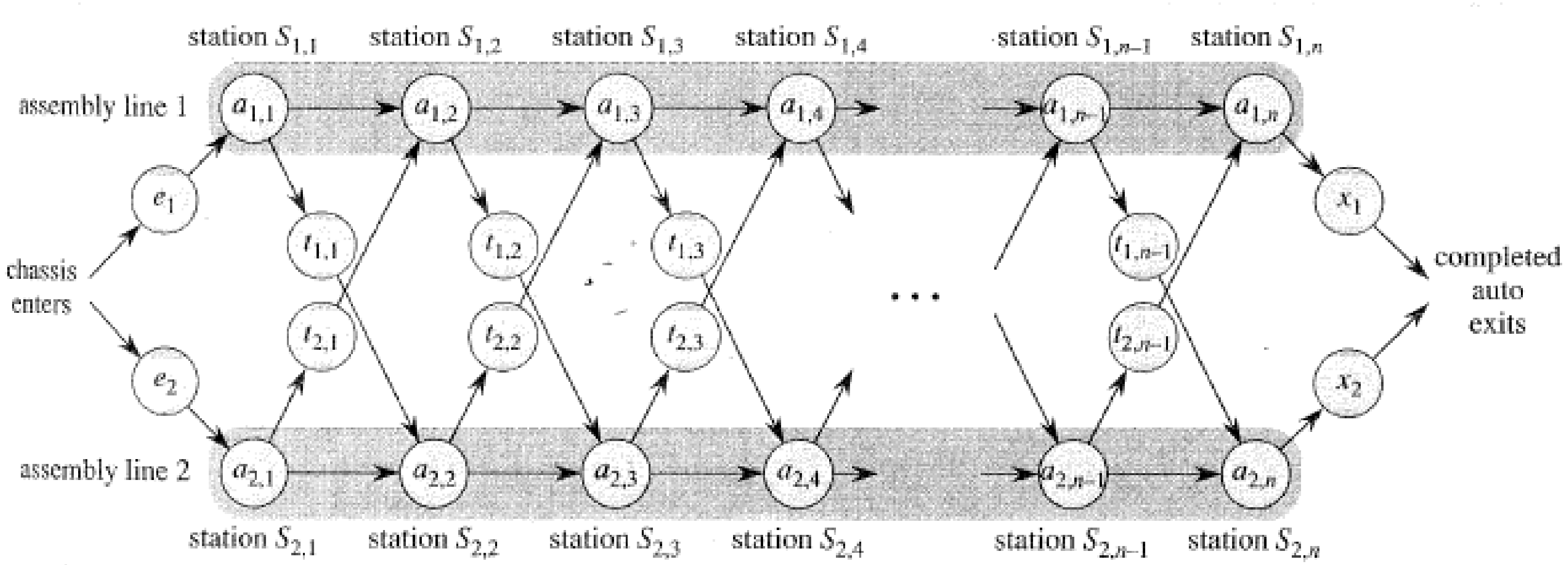
- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Dynamic Programming Algorithm

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information (not always necessary)

Assembly Line Scheduling

- Automobile factory with two assembly lines
- Each line has n stations: S1,1, . . . , S1,n and S2,1, . . . , S2,n
- Corresponding stations S1, j and S2, j perform the same function but can take different amounts of time a1, j and a2, j
- Entry times are: e1 and e2; exit times are: x1 and x2

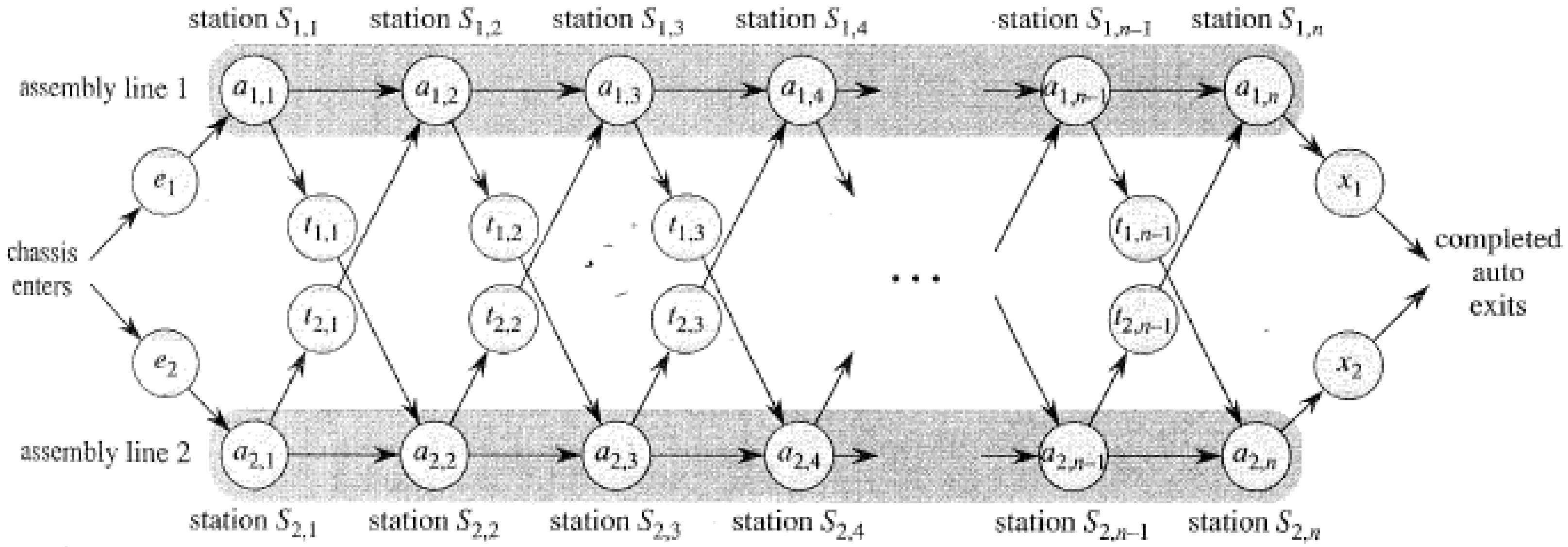


Assembly Line Scheduling

- After going through a station, can either:

  stay on same line at no cost, or

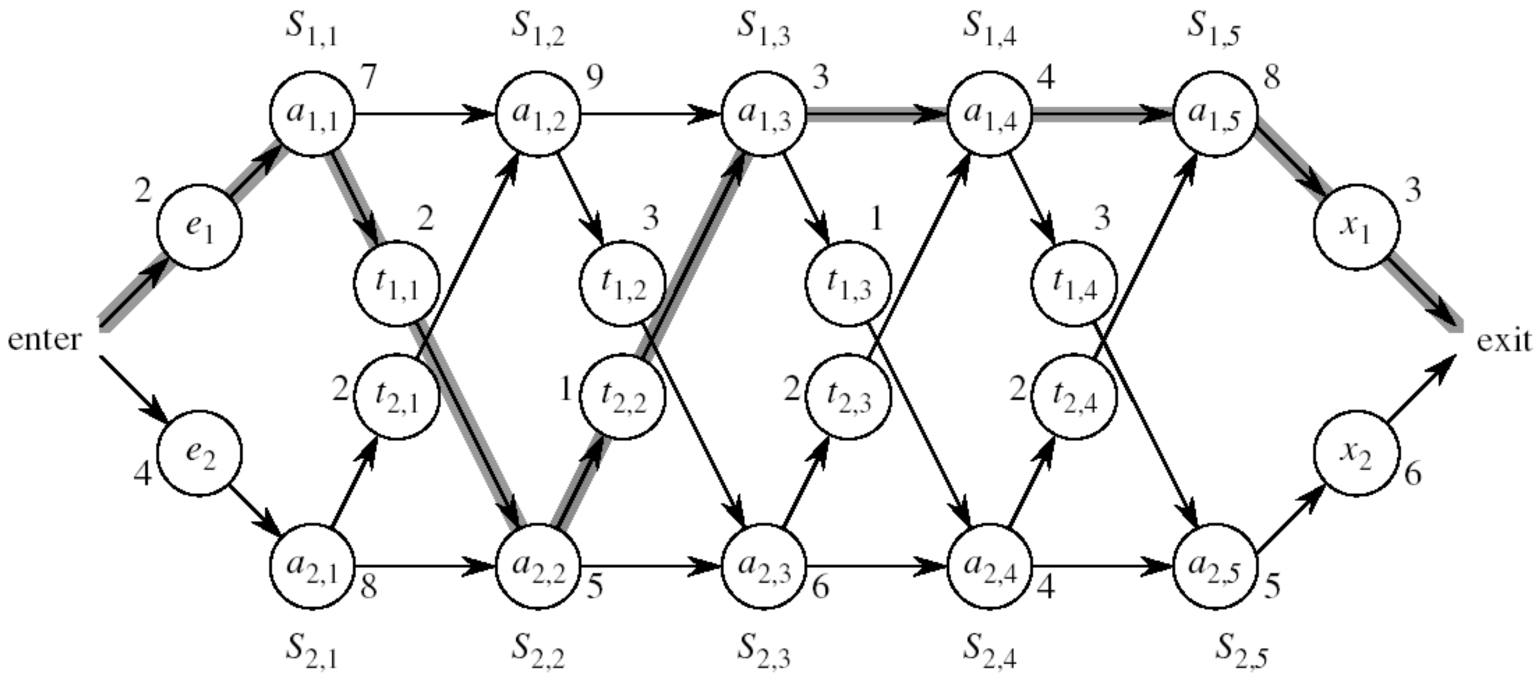
- transfer to other line: cost after Si,j is ti,j , j = 1, . . . , n - 1



Assembly Line Scheduling

Problem:

what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?

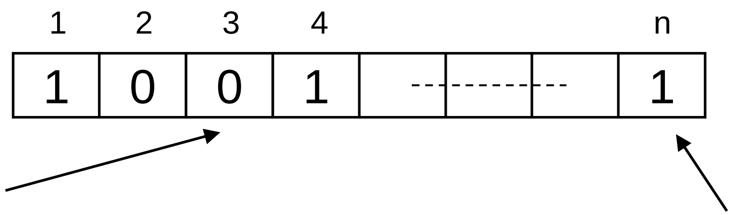


One Solution

Brute force

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

Solution:



1	2	3	4				n
1	0	0	1	----	----------	---	1

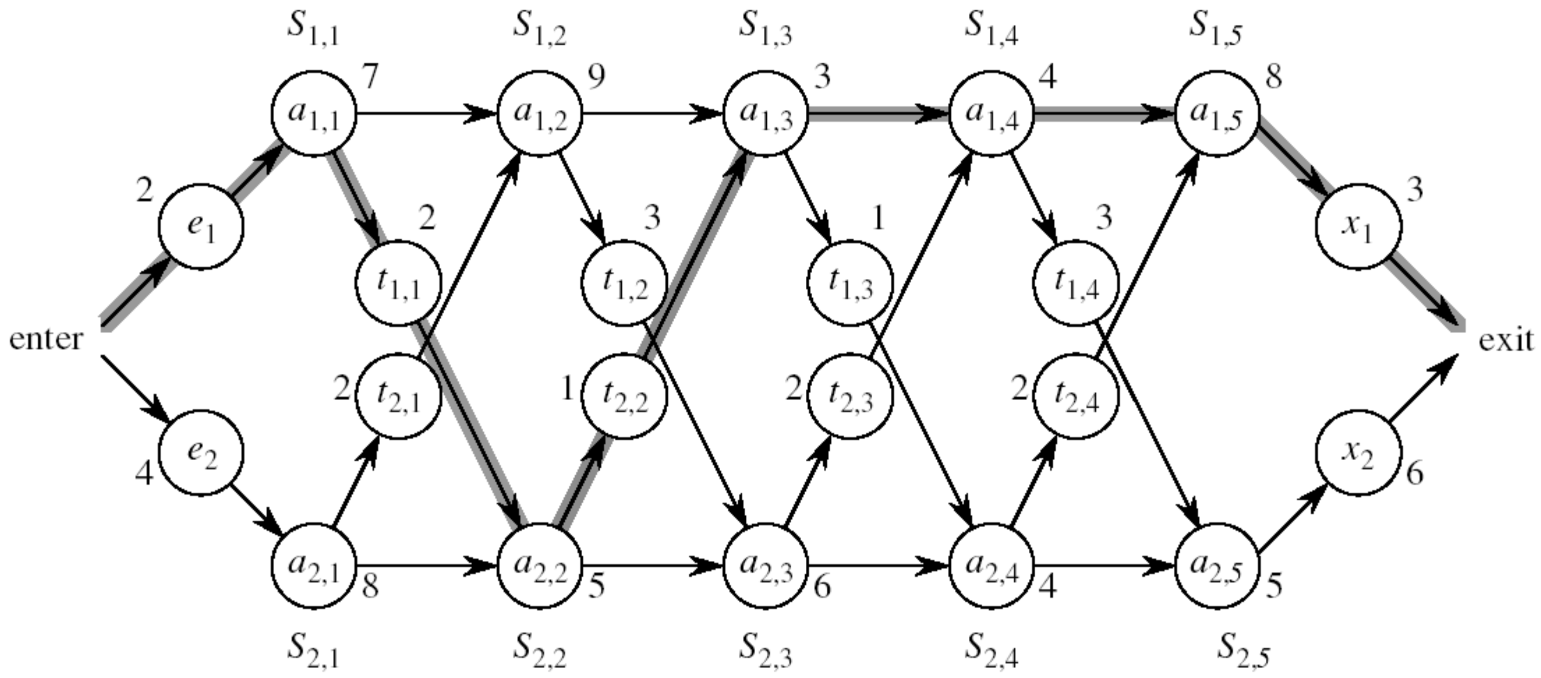
0 if choosing line 2
at step j (= 3)

1 if choosing line 1
at step j (= n)

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

1. Structure of the Optimal Solution

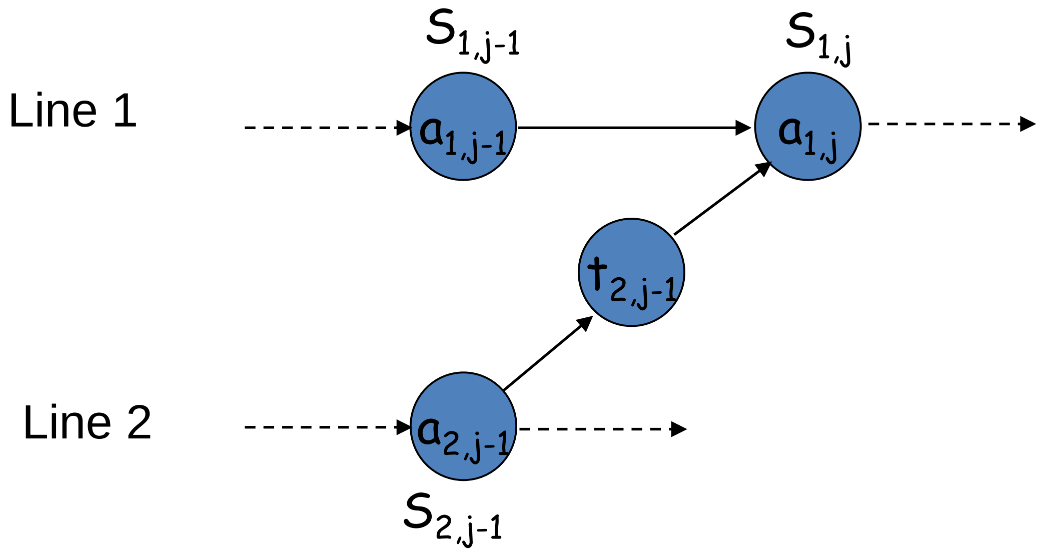
How do we compute the minimum time of going through a station?



1. Structure of the Optimal Solution

Let's consider all possible ways to get from the starting point through station S1,j

- We have two choices of how to get to S1, j:
  - Through S1, j - 1, then directly to S1, j
  - Through S2, j - 1, then transfer over to S1, j



1. Structure of the Optimal Solution

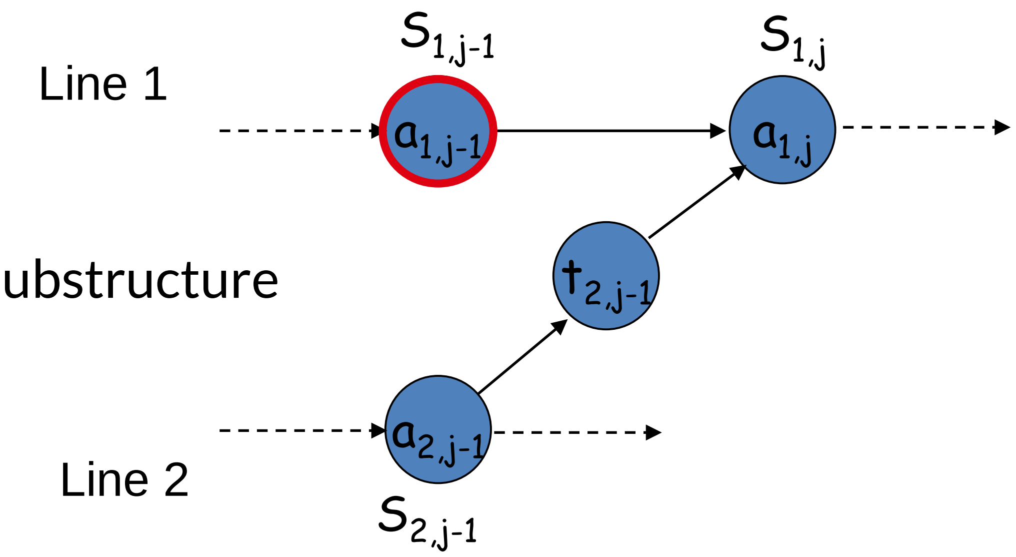
Suppose that the fastest way through $S_{1,j}$ is through $S_{1,j-1}$

We must have taken a fastest way from entry through $S_{1,j-1}$

If there were a faster way through $S_{1,j-1}$, we would use it instead

Similarly for $S_{2,j-1}$

Optimal Substructure



Optimal Substructure

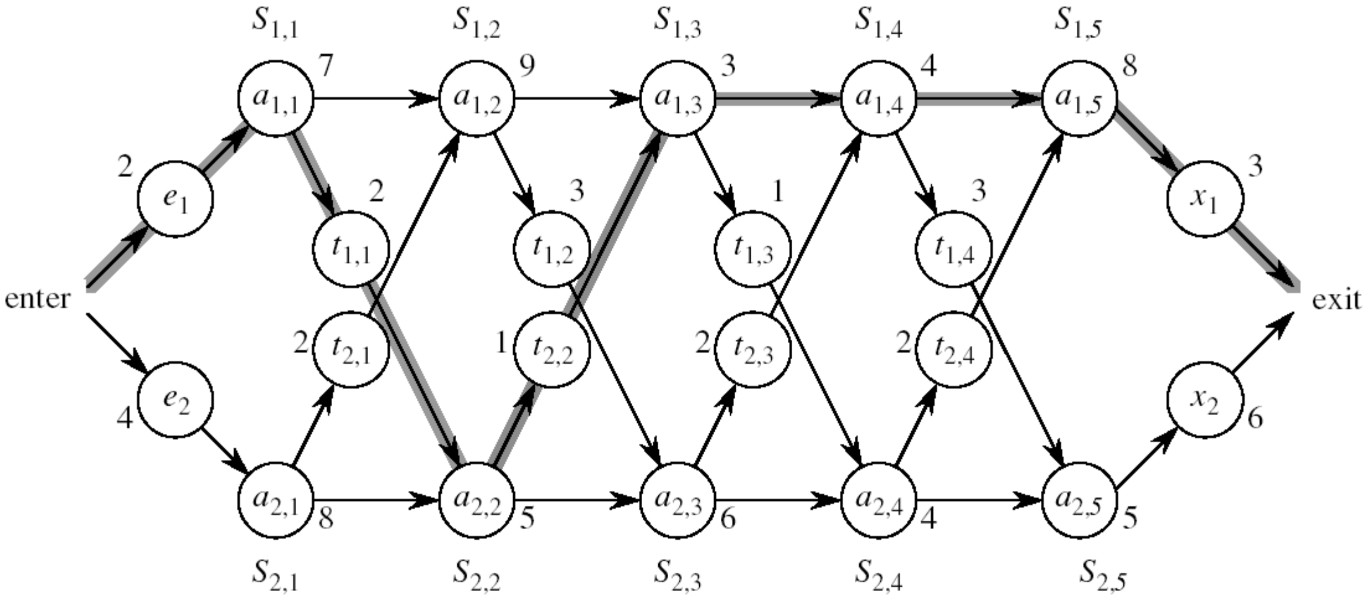
Generalization: an optimal solution to the problem "find the fastest way through $S_{1,j}$" contains within it an optimal solution to subproblems: "find the fastest way through $S_{1,j-1}$ or $S_{2,j-1}$".

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

2. A Recursive Solution

Define the value of an optimal solution in terms of the optimal solution to subproblems



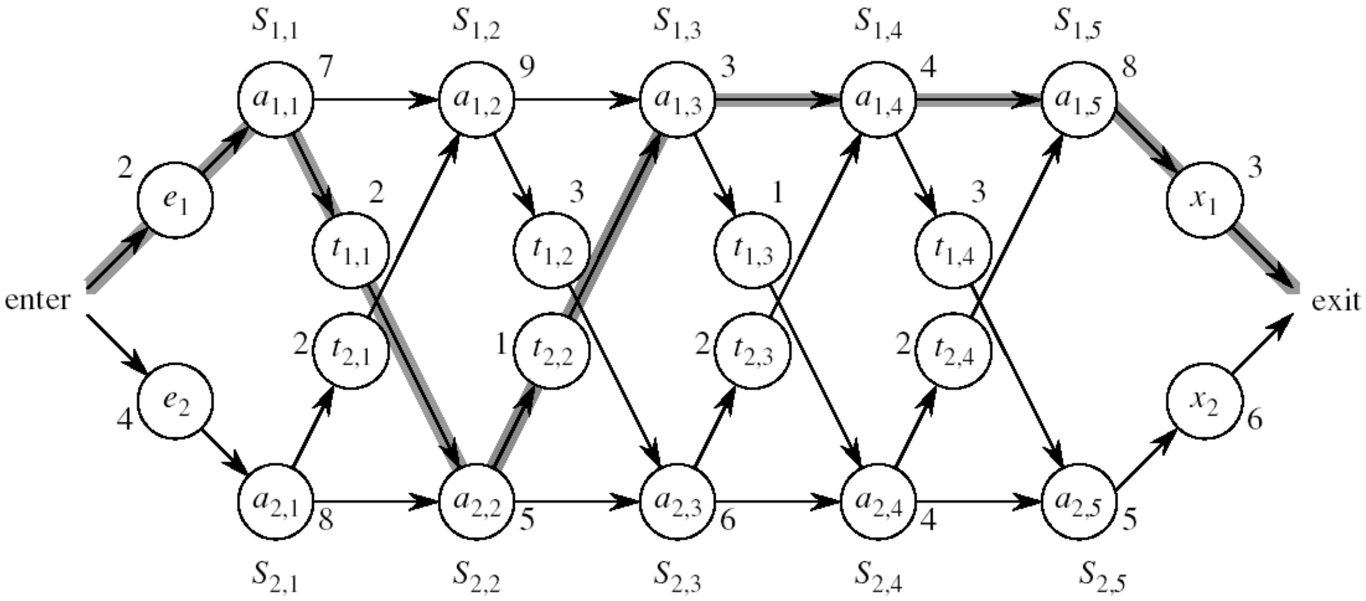
2. A Recursive Solution (cont.)

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 $S_{i,j}$

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

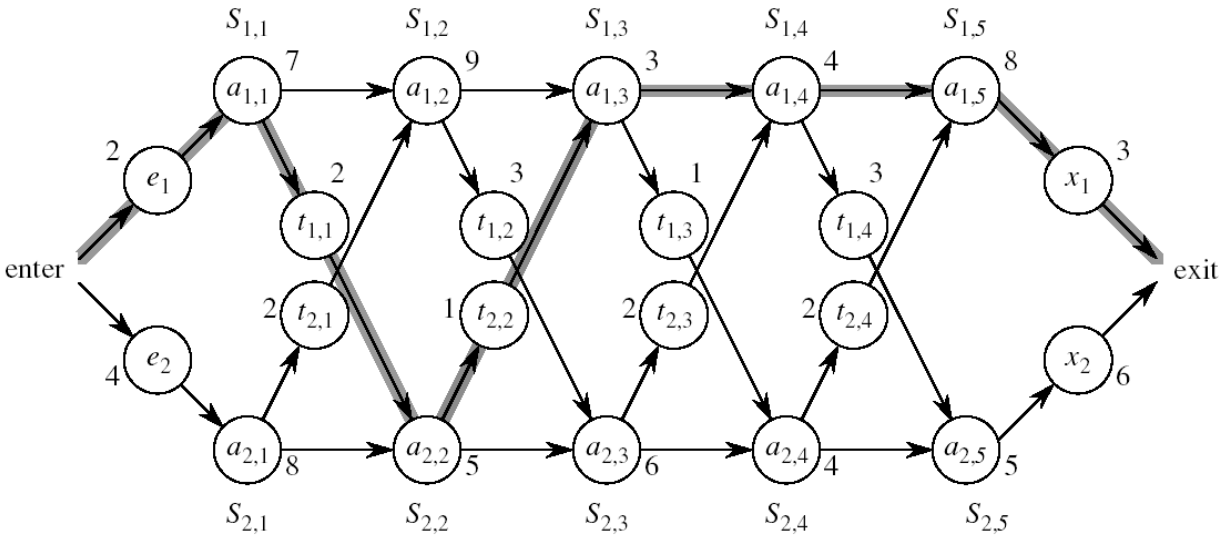


2. A Recursive Solution (cont.)

Base case: j = 1, i=1,2 (getting through station 1)

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

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



2. A Recursive Solution (cont.)

General Case: j = 2, 3, …,n, and i = 1, 2

Fastest way through $S_{1,j}$ is either:

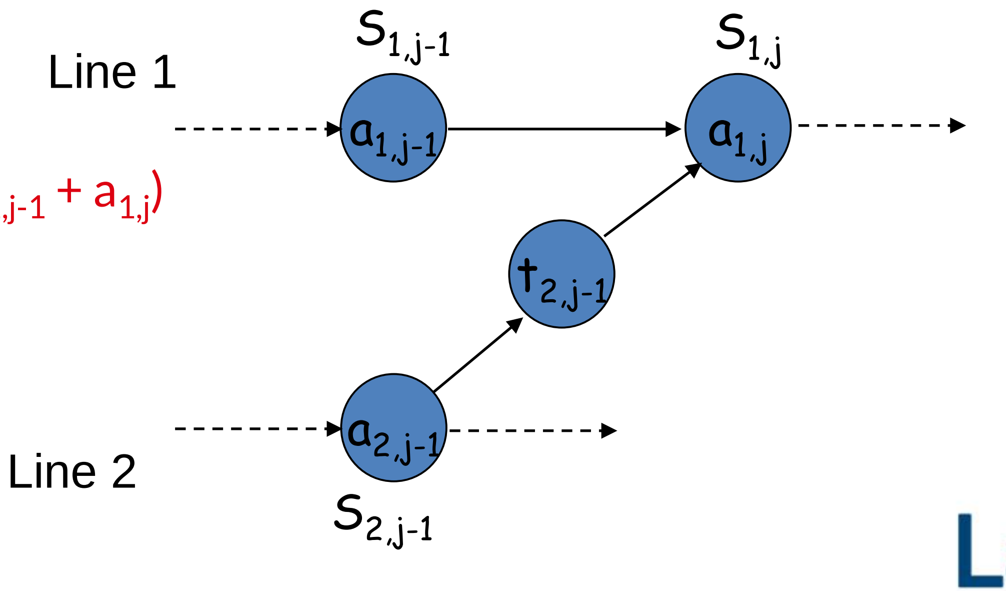
the way through $S_{1,j-1}$ then directly through $S_{1,j}$, or

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

the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$

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

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



2. A Recursive Solution (cont.)

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

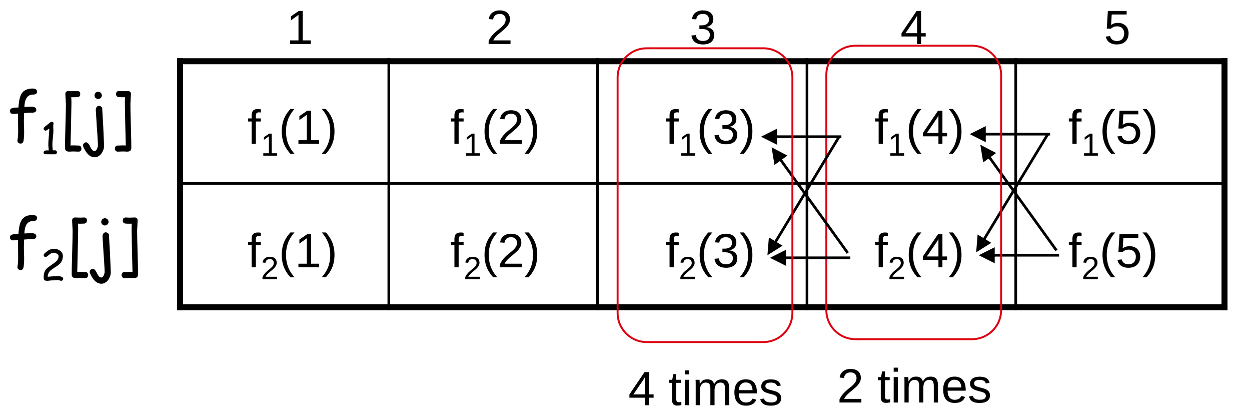
$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j - 1] + a_{2,j} ,f_1[j -1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

3. Computing the Optimal Solution

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

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

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



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

4 times

2 times

Solving top-down would result in exponential running time

3. Computing the Optimal Solution

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:

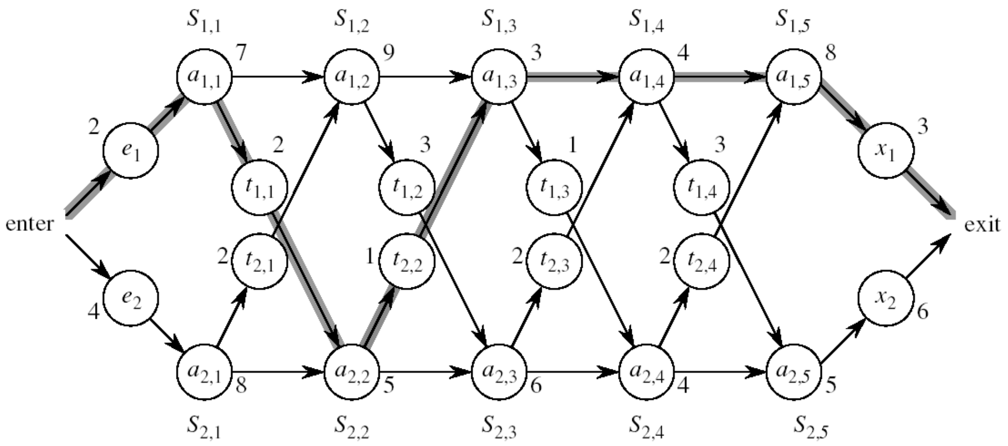
in increasing order of j

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

Bottom-up approach

- First find optimal solutions to subproblems
- Find an optimal solution to the problem from the subproblems

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
$f_1[j]$	9	18[1]	20[2]	24[1]	32[1]
$f_2[j]$	12	16[1]	22[2]	25[1]	30[2]

f* = 35[1]

FASTEST-WAY(a, t, e, x, n)

```
$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ to $n$
    do if $f_1[j - 1] + a_{1,j} \le f_2[j - 1] + t_{2, j-1} + a_{1, j}$
        then $f_1[j] \leftarrow f_1[j - 1] + a_{1, j}$
            $l_1[j] \leftarrow 1$
        else $f_1[j] \leftarrow f_2[j - 1] + t_{2, j-1} + a_{1, j}$
            $l_1[j] \leftarrow 2$
    if $f_2[j - 1] + a_{2, j} \le f_1[j - 1] + t_{1, j-1} + a_{2, j}$
        then $f_2[j] \leftarrow f_2[j - 1] + a_{2, j}$
            $l_2[j] \leftarrow 2$
        else $f_2[j] \leftarrow f_1[j - 1] + t_{1, j-1} + a_{2, j}$
            $l_2[j] \leftarrow 1$
```

Compute initial values of f_1 and f_2

O(N)

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

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

FASTEST-WAY(a, t, e, x, n) (cont.)

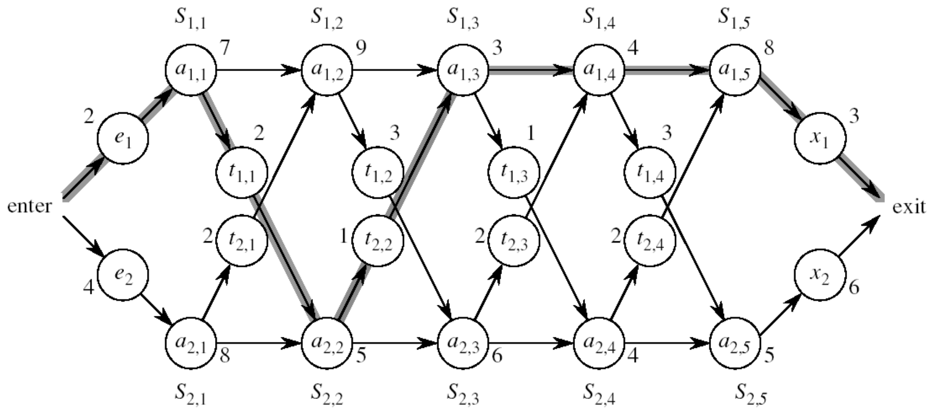
```
if $f_1[n] + x_1 \leq f_2[n] + x_2$
    then $f^* = f_1[n] + x_1$
        $l^* = 1$
    else $f^* = f_2[n] + x_2$
        $l^* = 2$
```

Compute the values of the fastest time through the entire factory

4. Construct an Optimal Solution

Alg.: 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
$f_1[j]/l_1[j]$	9	18[1]	20[2]	24[1]	32[1]
$f_2[j]/l_2[j]$	12	16[1]	22[2]	25[1]	30[2]

l* = 1

Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

Matrix compatibility:

$C = A \cdot B$

$\text{col}_A = \text{row}_B$

$\text{row}_C = \text{row}_A$

$\text{col}_C = \text{col}_B$

$C=A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$

$\text{col}_i = \text{row}_{i+1}$

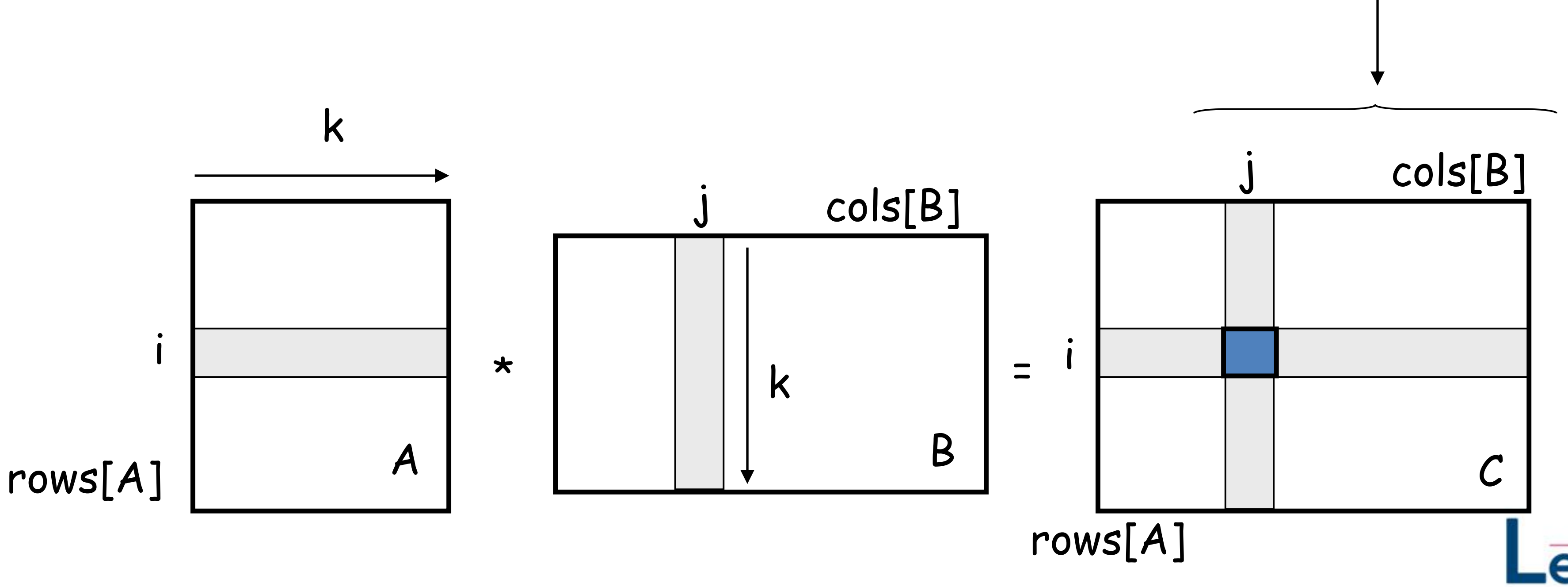
$\text{row}_C = \text{row}_{A1}$

$\text{col}_C = \text{col}_{An}$

MATRIX-MULTIPLY(A, B)

```
if columns[A] $\neq$ rows[B]
    then error "incompatible dimensions"
    else for i $\leftarrow$ 1 to rows[A]
            do for j $\leftarrow$ 1 to columns[B]
                do $C[i, j] = 0$
                    for k $\leftarrow$ 1 to columns[A]
                        do $C[i, j] \leftarrow C[i, j] + A[i, k]\, B[k, j]$
```

rows[A] · cols[A] · cols[B]
multiplications



Matrix-Chain Multiplication

In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

Parenthesize the product to get the order in which matrices are multiplied

E.g.: $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$= (A_1 \cdot (A_2 \cdot A_3))$

Which one of these orderings should we choose?

The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$A_1 \cdot A_2 \cdot A_3$

A_1: 10 x 100

A_2: 100 x 5

A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2$ = 10 x 100 x 5 = 5,000 (10 x 5)

$((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500

Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3$ = 100 x 5 x 50 = 25,000 (100 x 50)

$(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000

Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$\begin{array}{ccccccccc} A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\ p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n \end{array}$$

What is the number of possible parenthesizations?

Exhaustively checking all possible parenthesizations is not efficient!

It can be shown that the number of parenthesizations grows as $\Omega(4n/n3/2)$

(see page 333 in your textbook)

1. The Structure of an Optimal Parenthesization

Notation:

$$A_{i...j} = A_i\ A_{i+1} \cdots A_j,\ i \leq j$$

Suppose that an optimal parenthesization of $A_{i...j}$ splits the product between A_k and A_{k+1}, where $i \leq k < j$

$$\begin{aligned} A_{i...j} &= A_i\ A_{i+1} \cdots A_j \\ &= A_i\ A_{i+1} \cdots A_k\ A_{k+1} \cdots A_j \\ &= A_{i...k}\ A_{k+1...j} \end{aligned}$$

Optimal Substructure

$$A_{i\ldots j} = A_{i\ldots k} A_{k+1\ldots j}$$

The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parentesization

If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum ⇒ contradiction!

An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

2. A Recursive Solution

Subproblem:

determine the minimum cost of parenthesizing $A_{i...j} = A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$

Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i...j}$

full problem ($A_{1..n}$): $m[1, n]$

$i = j$: $A_{i...i} = A_i \Rightarrow m[i, i] =$

0, for i = 1, 2, …, n

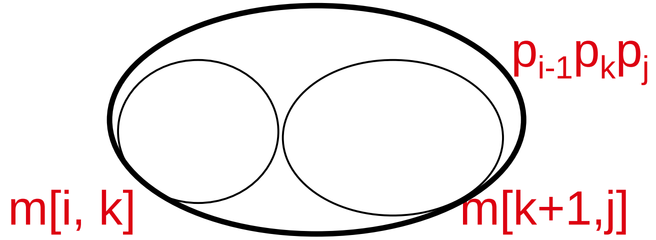
2. A Recursive Solution

Consider the subproblem of parenthesizing $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

$= A_{i\ldots k} A_{k+1\ldots j}$ for $i \le k < j$

Assume that the optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ at k ($i \le k < j$)

$m[i, j] =$



$$\underbrace{m[i, k]}_{\substack{\text{min \# of multiplications} \\ \text{to compute } A_{i\ldots k}}} + \underbrace{m[k+1, j]}_{\substack{\text{min \# of multiplications} \\ \text{to compute } A_{k+1\ldots j}}} + \underbrace{p_{i-1}p_kp_j}_{\substack{\text{\# of multiplications} \\ \text{to compute } A_{i\ldots k}A_{k\ldots j}}}$$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] \quad + \quad m[k+1, j] \quad + \quad p_{i-1}p_kp_j$$

We do not know the value of k

There are j – i possible values for k: k = i, i+1, …, j-1

Minimizing the cost of parenthesizing the product $A_i\ A_{i+1} \cdots A_j$ becomes:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

3. Computing the Optimal Costs

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Computing the optimal solution recursively takes exponential time!

How many subproblems?

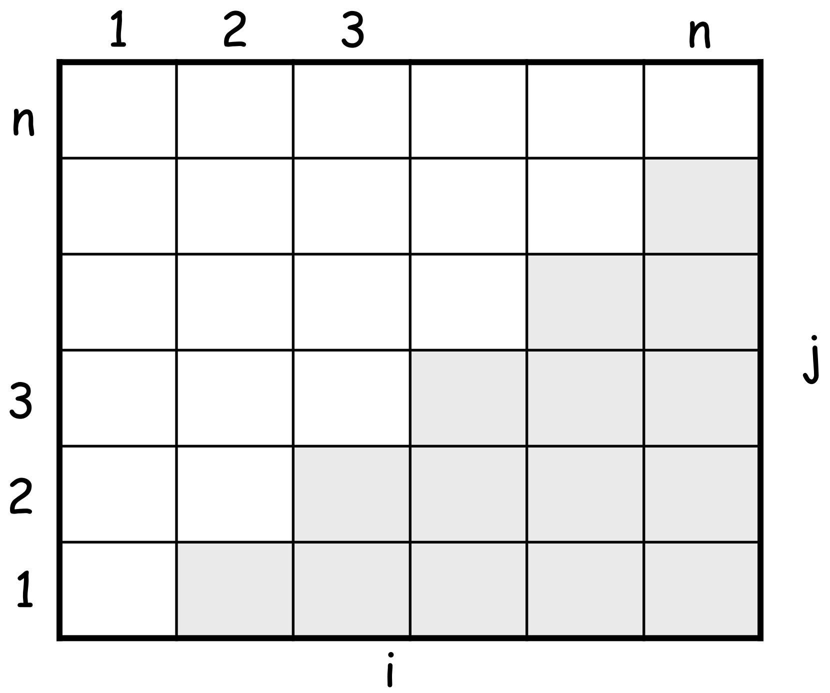
Parenthesize $A_{i…j}$

for $1 \le i \le j \le n$

One problem for each

choice of i and j

$\Rightarrow \Theta(n^2)$



3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

How do we fill in the tables m[1..n, 1..n]?

Determine which entries of the table are used in computing m[i, j]

$$A_{i...j} = A_{i...k} A_{k+1...j}$$

Subproblems' size is one less than the original size

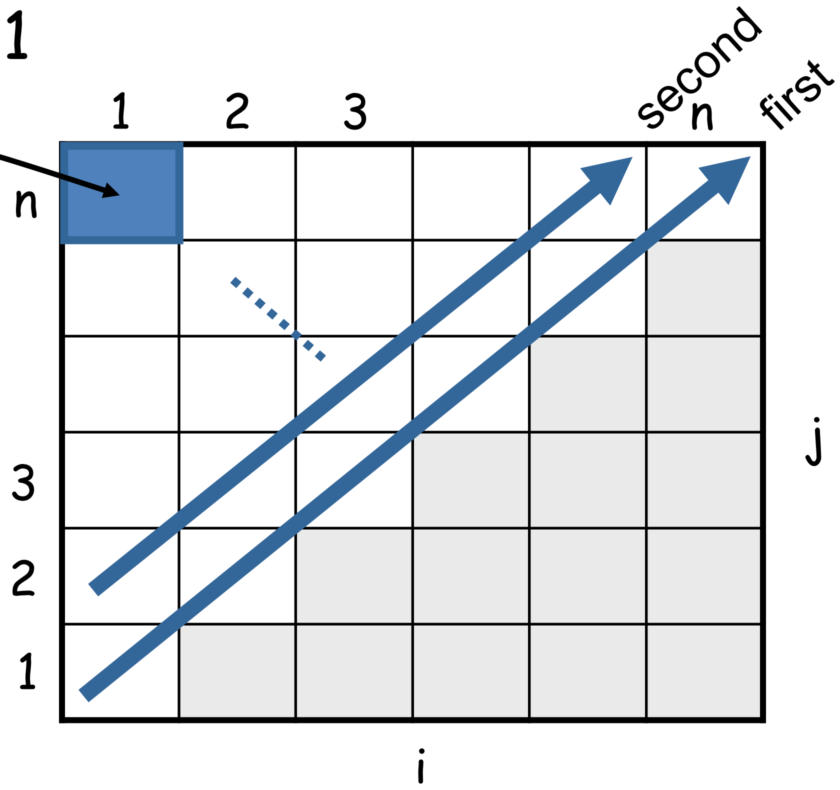
Idea: fill in m such that it corresponds to solving problems of increasing length

3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Length = 1: $i = j$, $i = 1, 2, \ldots, n$

Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

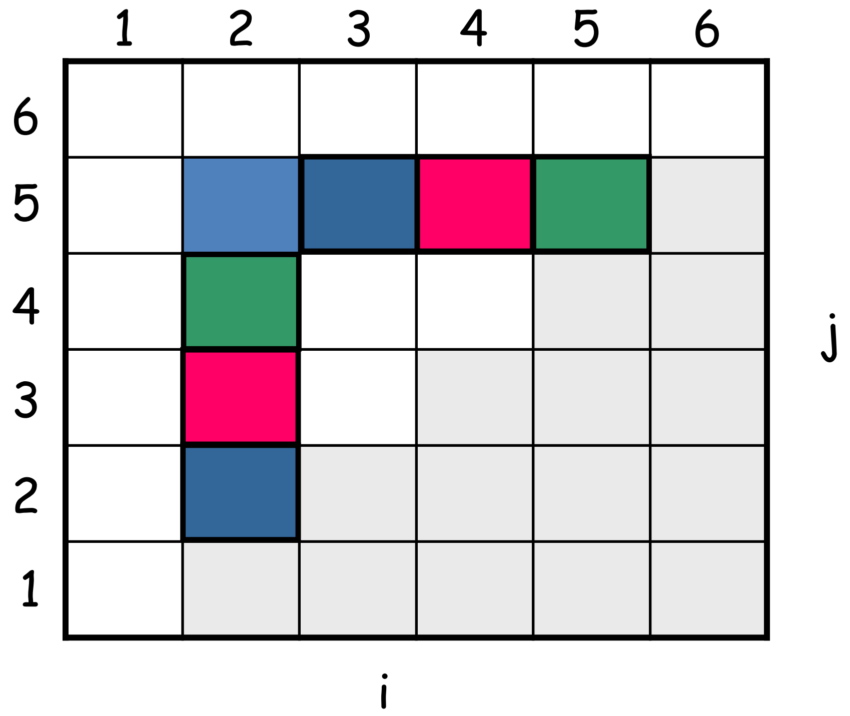


m[1, n] gives the optimal solution to the problem

Compute rows from bottom to top and from left to right

Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$



- Values m[i, j] depend only on values that have been previously computed

Example: min {m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$}

Compute $A_1 \cdot A_2 \cdot A_3$

A_1: 10 x 100 (p_0 x p_1)

A_2: 100 x 5   (p_1 x p_2)

A_3: 5 x 50   (p_2 x p_3)

m[i, i] = 0 for i = 1, 2, 3

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

m[1, 2] = m[1, 1] + m[2, 2] + $p_0p_1p_2$ (A_1A_2)

= 0 + 0 + 10 *100* 5 = 5,000

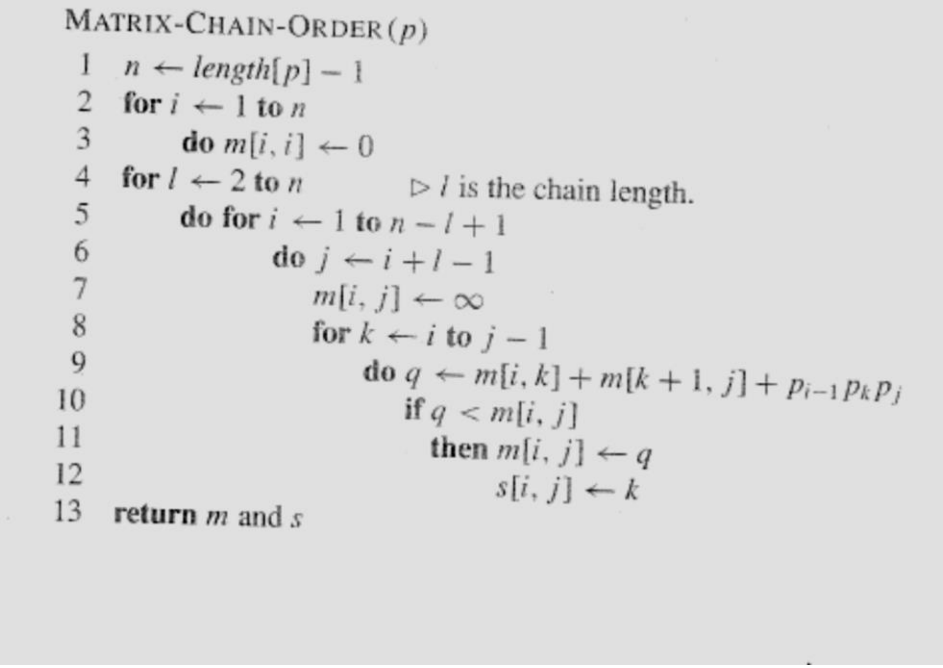
m[2, 3] = m[2, 2] + m[3, 3] + $p_1p_2p_3$ (A_2A_3)

= 0 + 0 + 100 * 5 * 50 = 25,000

m[1, 3] = min m[1, 1] + m[2, 3] + $p_0p_1p_3$ = 75,000 ($A_1(A_2A_3)$)

m[1, 2] + m[3, 3] + $p_0p_2p_3$ = 7,500   ($(A_1A_2)A_3$)

Matrix-Chain-Order(p)



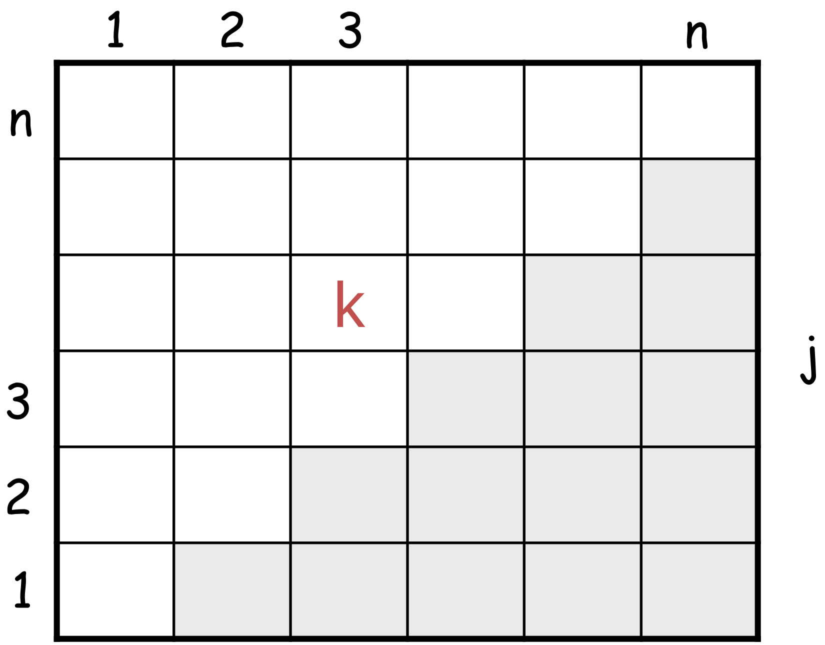
```
MATRIX-CHAIN-ORDER($p$)
$n \leftarrow length[p] - 1$
for $i \leftarrow 1$ to $n$
    do $m[i, i] \leftarrow 0$
for $l \leftarrow 2$ to $n$                    ▷ $l$ is the chain length.
    do for $i \leftarrow 1$ to $n - l + 1$
           do $j \leftarrow i + l - 1$
              $m[i, j] \leftarrow \infty$
              for $k \leftarrow i$ to $j - 1$
                  do $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
                     if $q < m[i, j]$
                       then $m[i, j] \leftarrow q$
                            $s[i, j] \leftarrow k$
return $m$ and $s$
```

$O(N^3)$

4. Construct the Optimal Solution

In a similar matrix s we keep the optimal values of k

$s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}



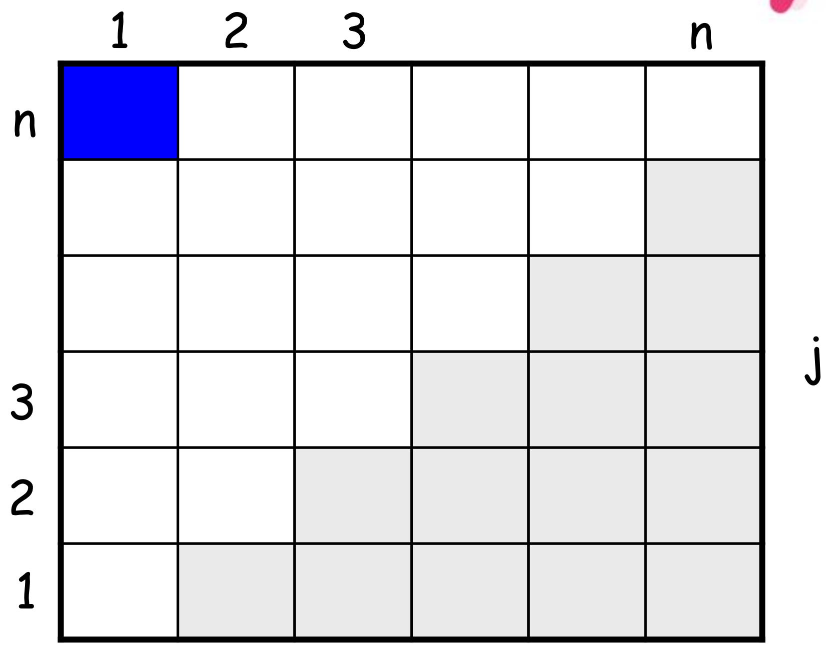
4. Construct the Optimal Solution

$s[1, n]$ is associated with the entire product $A_{1..n}$

- The final matrix multiplication will be split at k = $s[1, n]$

$$A_{1..n} = A_{1..s[1,\, n]} \cdot A_{s[1,\, n]+1..n}$$

- For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



4. Construct the Optimal Solution

s[i, j] = value of k such that the optimal parenthesization of $A_i\ A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\ A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\ A_{6..6}$

4. Construct the Optimal Solution (cont.)

```
PRINT-OPT-PARENS(s, i, j)
if i = j
  then print “A”$_i$
  else print “(”
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print “)”
```

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

Example: A1· · ·A6

$((A_1(A_2A_3))((A_4A_5)A_6))$

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"$_i$
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print ")"
```

s[1..6, 1..6]

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) ⇒ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) ⇒ "A_2"

P-O-P (s, 3, 3) ⇒ "A_3"

")"

")" …

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem

  memoize the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent "calls" to the subproblem simply look up that value

Memoized Matrix-Chain

Alg.: MEMOIZED-MATRIX-CHAIN(p)

```
n ← length[p] - 1
for i ← 1 to n
        do for j ← i to n
                do m[i, j] ← $\infty$
return LOOKUP-CHAIN(p, 1, n)
```

Initialize the m table with large values that indicate whether the values of m[i, j] have been computed

← Top-down approach

Memoized Matrix-Chain

Alg.: LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

```
if m[i, j] < ∞
    then return m[i, j]
if i = j
  then m[i, j] ← 0
  else for k ← i to j - 1
          do q ← LOOKUP-CHAIN(p, i, k) +
                 LOOKUP-CHAIN(p, k+1, j) + $p_{i-1}p_kp_j$
             if q < m[i, j]
                then m[i, j] ← q
return m[i, j]
```

Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms

  No overhead for recursion, less overhead for maintaining the table

  The regular pattern of table accesses may be used to reduce time or space requirements

- Advantages of memoized algorithms vs. dynamic programming

  Some subproblems do not need to be solved

Elements of Dynamic Programming

- Optimal Substructure

  An optimal solution to a problem contains within it an optimal solution to subproblems

  Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems

- Overlapping Subproblems

  If a recursive algorithm revisits the same subproblems over and over ⇒ the problem has overlapping subproblems

Parameters of Optimal Substructure

- How many subproblems are used in an optimal solution for the original problem

Assembly line: One subproblem (the line that gives best time)

Matrix multiplication: Two subproblems (subproducts $A_{i..k}$, $A_{k+1..j}$)

- How many choices we have in determining which subproblems to use in an optimal solution

Assembly line: Two choices (line 1 or line 2)

Matrix multiplication: $j - i$ choices for k (splitting the product)

Parameters of Optimal Substructure

Intuitively, the running time of a dynamic programming algorithm depends on two factors:

- Number of subproblems overall
- How many choices we look at for each subproblem

Assembly line

- $\Theta(n)$ subproblems (n stations)
- 2 choices for each subproblem

$\Theta(n)$ overall

Matrix multiplication:

- Θ(n2) subproblems ($1 \leq i \leq j \leq n$)
- At most $n-1$ choices

$\Theta(n^3)$ overall

Longest Common Subsequence

Given two sequences

$X = \langle x_1, x_2, \ldots, x_m \rangle$

$Y = \langle y_1, y_2, \ldots, y_n \rangle$

find a maximum length common subsequence (LCS) of X and Y

E.g.:

$X = \langle A, B, C, B, D, A, B \rangle$

Subsequences of X:

A subset of elements in the sequence taken in order

$\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example

X = ⟨A, B, C, B, D, A, B⟩ X = ⟨A, B, C, B, D, A, B⟩

Y = ⟨B, D, C, A, B, A⟩ Y = ⟨B, D, C, A, B, A⟩

⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are longest common subsequences of X and Y (length = 4)

⟨B, C, A⟩, however is not a LCS of X and Y

Brute-Force Solution

For every subsequence of X, check whether it’s a subsequence of Y

There are 2^m subsequences of X to check

Each subsequence takes $\Theta(n)$ time to check

  scan Y for first letter, from there scan for second, and so on

Running time: $\Theta(n2^m)$

Making the choice

X = ⟨A, B, D, E⟩

Y = ⟨Z, B, E⟩

Choice: include one element into the common sequence (E) and solve the resulting subproblem

X = ⟨A, B, D, G⟩

Y = ⟨Z, B, D⟩

Choice: exclude an element from a string and solve the resulting subproblem

Notations

Given a sequence $X = \langle x1, x2, \ldots, xm \rangle$ we define the i-th prefix of X, for $i = 0, 1, 2, \ldots, m$

$$Xi = \langle x1, x2, \ldots, xi \rangle$$

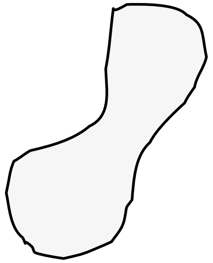
$c[i, j]$ = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

A Recursive Solution

Case 1: xi = yj

e.g.: Xi = ⟨A, B, D, E⟩

Yj = ⟨Z, B, E⟩



$$c[i, j] = c[i - 1, j - 1] + 1$$

Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}

Must find a LCS of X_{i-1} and Y_{j-1} ⇒ optimal solution to a problem includes optimal solutions to subproblems

A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i - 1, j], c[i, j-1] \}$$

Must solve two problems

find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$

find a LCS of X_i and Y_{j-1}: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

Optimal solution to a problem includes optimal solutions to subproblems

Overlapping Subproblems

To find a LCS of X and Y

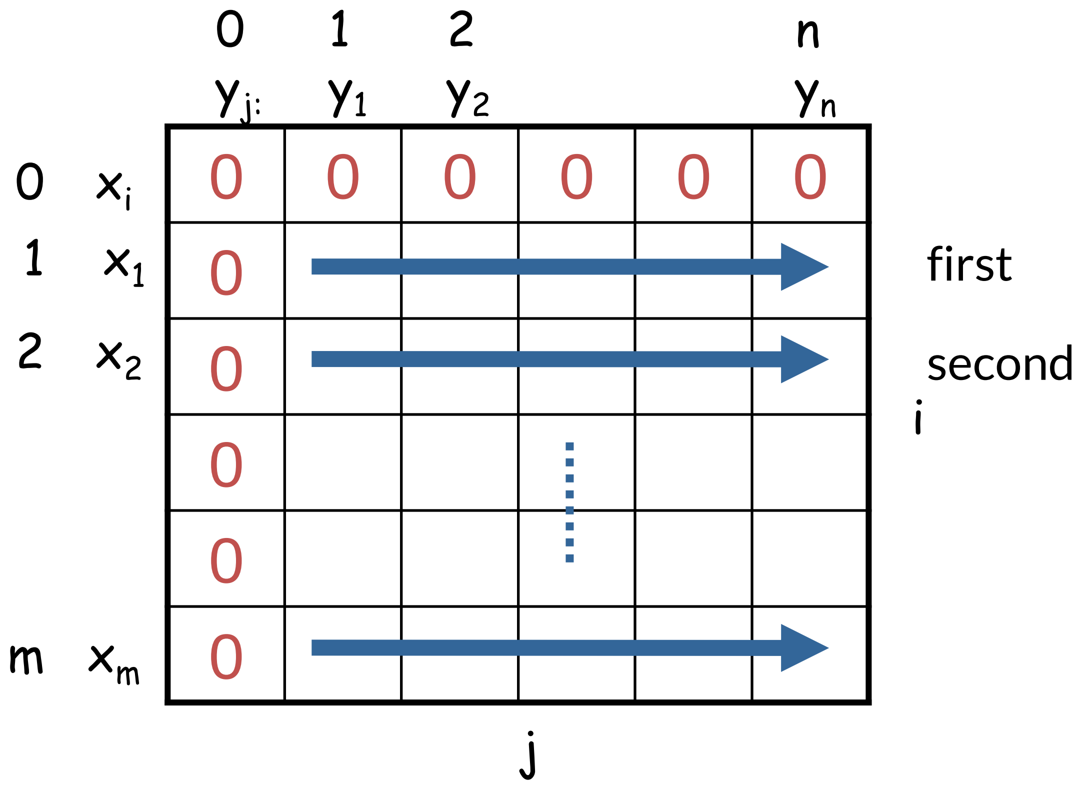
we may need to find the LCS between X and Y_{n-1} and that of X_{m-1} and Y

Both the above subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}

Subproblems share subsubproblems

3. Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

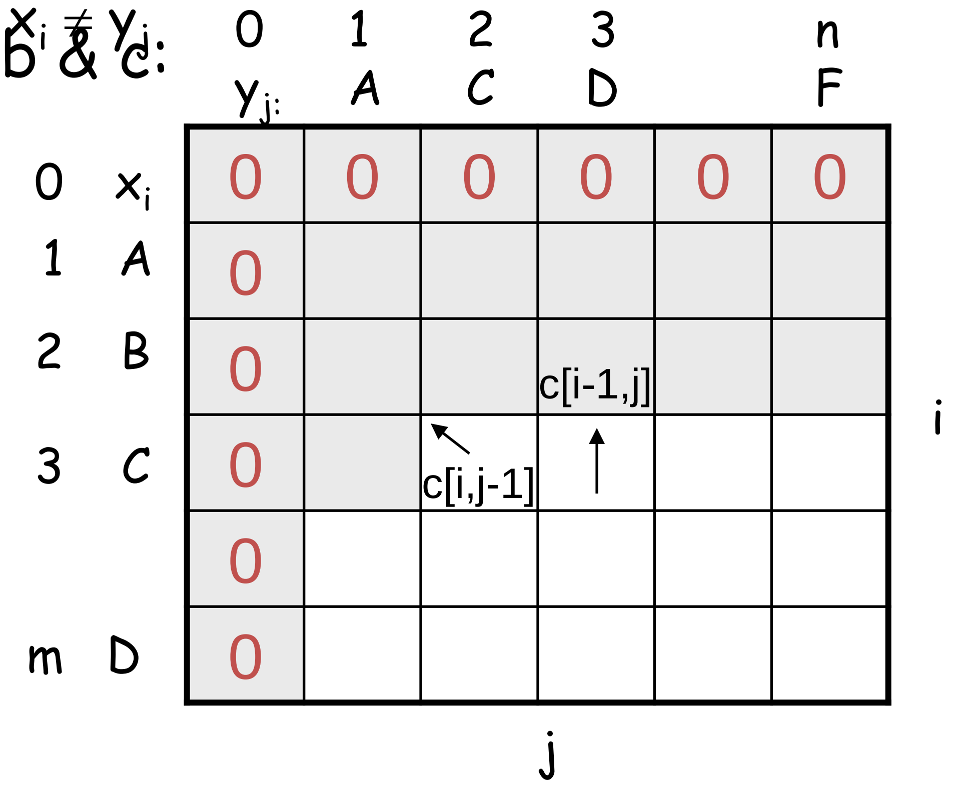


		0 $y_{j:}$	1 y_1	2 y_2			n y_n
0	x_i	0	0	0	0	0	0
1	x_1	0					
2	x_2	0					
		0					
		0					
m	x_m	0					

Additional Information

$$c[i, j] = \begin{cases} 0 & \text{if } i,j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

b & c:



		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$

  b[i, j] = “↖”
- Else, if c[i - 1, j] ≥ c[i, j-1]

  b[i, j] = “ ↑ ”

  else

  b[i, j] = “ ← ”

LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if $x_i = y_j$
            then c[i, j] ← c[i - 1, j - 1] + 1
                b[i, j ] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j - 1]
                    b[i, j] ← "←"
return c and b
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

Example

$X = \langle A, B, C, B, D, A \rangle$
$Y = \langle B, D, C, A, B, A \rangle$

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

If $x_i = y_j$

$b[i, j] =$ " ↖ "

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] =$ " ↑ "

else

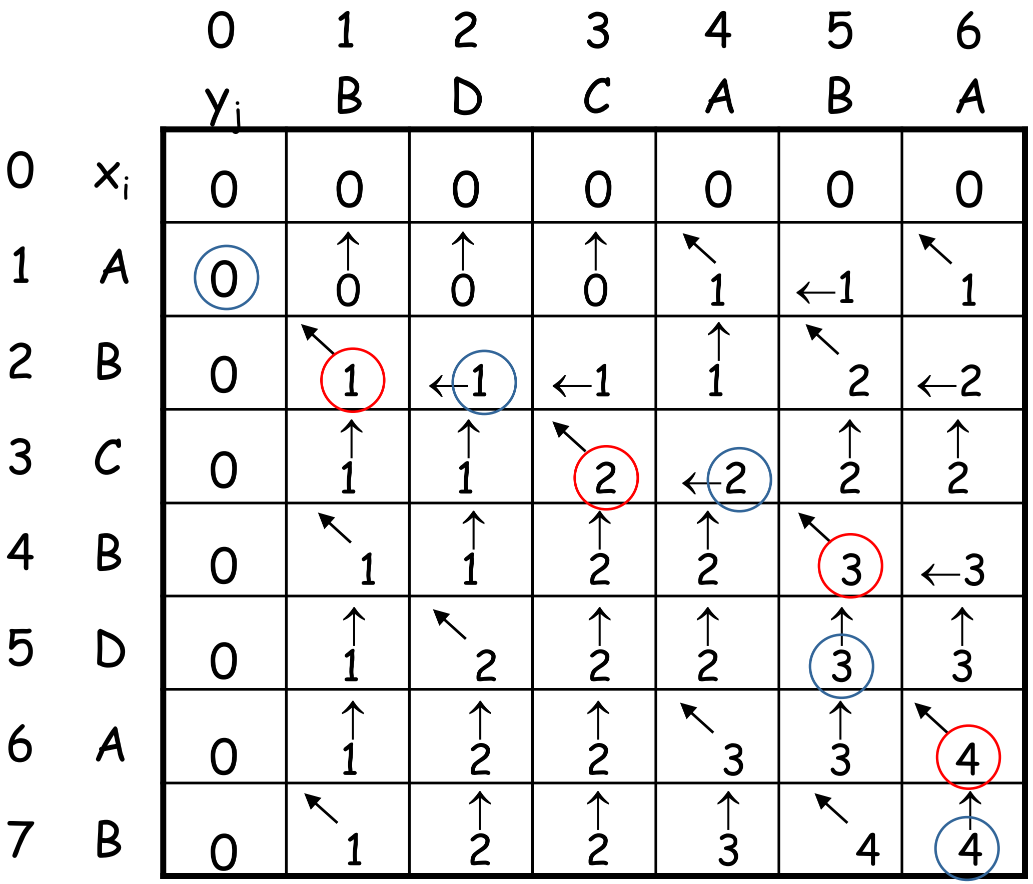
$b[i, j] =$ " ← "

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

4. Constructing a LCS

Start at b[m, n] and follow the arrows

When we encounter a “↖“ in b[i, j] ⇒ $x_i = y_j$ is an element of the LCS



		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

PRINT-LCS(b, X, i, j)

```
if i = 0 or j = 0
  then return
if b[i, j] = “   ”
    then PRINT-LCS(b, X, i - 1, j - 1)
         print $x_i$
elseif b[i, j] = “↑”
         then PRINT-LCS(b, X, i - 1, j)
         else PRINT-LCS(b, X, i, j - 1)
```

Running time: $\Theta(m + n)$

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Improving the Code

What can we say about how each entry $c[i, j]$ is computed?

- It depends only on $c[i -1, j - 1]$, $c[i - 1, j]$, and $c[i, j - 1]$
- Eliminate table b and compute in $O(1)$ which of the three values was used to compute $c[i, j]$
- We save $\Theta(mn)$ space from table b
- However, we do not asymptotically decrease the auxiliary space requirements: still need table c

Improving the Code

If we only need the length of the LCS

- LCS-LENGTH works only on two rows of c at a time
  - **The row being computed and the previous row**
- We can reduce the asymptotic space requirements by storing only these two rows

Dynamic programming

Problem: Let’s consider the calculation of **Fibonacci** numbers:

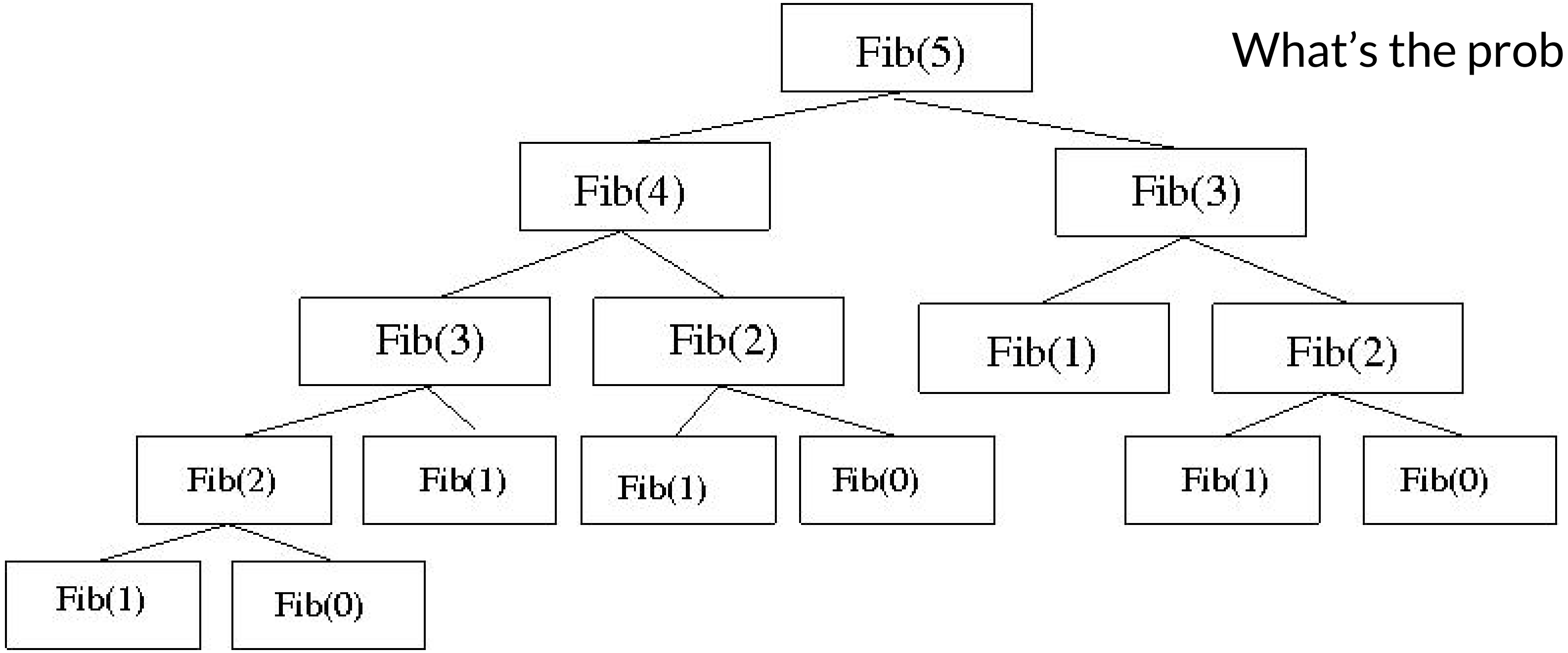
$F(n) = F(n\text{-}2) + F(n\text{-}1)$

with seed values $F(1) = 1, F(2) = 1$
or $F(0) = 0, F(1) = 1$

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, …

Dynamic programming



Fib(5)

Fib(4) Fib(3)

Fib(3) Fib(2) Fib(1) Fib(2)

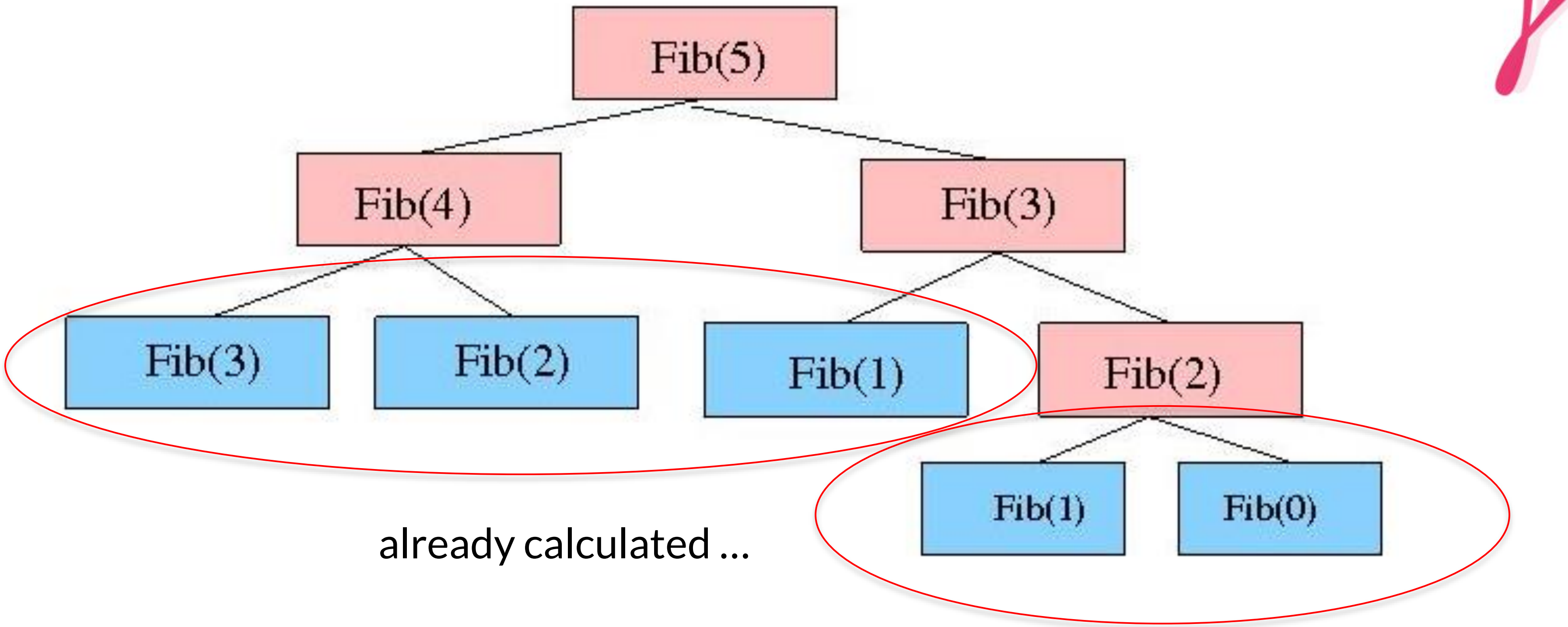
Fib(2) Fib(1) Fib(1) Fib(0) Fib(1) Fib(0)

Fib(1) Fib(0)

What’s the problem?

Memoization:

```
Fib(n)
{
   if (n == 0)
      return M[0];

   if (n == 1)
      return M[1];

   if (Fib(n-2) is not already calculated)
      call Fib(n-2);

   if(Fib(n-1) is already calculated)
      call Fib(n-1);

   //Store the ${n}^{th}$ Fibonacci no. in memory & use previous results.
   M[n] = M[n-1] + M[n-2]

   Return M[n];
}
```



already calculated …

Dynamic programming with rod cutting

- **Main approach:** recursive, holds answers to a sub problem in a table, can be used without recomputing.

- Can be formulated both via recursion and saving results in a table (*memoization*). Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.

- ”*programming*” as in tabular not programming code

Example: rod cutting

We are given prices p_i and rods of length i:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Question: We are given a rod of length n, and want to maximize revenue, by cutting up the rod into pieces and selling each of the pieces.

Example: we are given a 4 inches rod. Best solution to cut up? We'll first list the solutions:

We'll first list the solutions:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

1.) Cut into 2 pieces of length 2: $p_2 + p_2 = 5 + 5 = 10$

2.) Cut into 4 pieces of length 1: $p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$

3-4.) Cut into 2 pieces of length 1 and 3 (3 and 1): $p_1 + p_3 = 1 + 8 = 9$ $p_3 + p_1 = 8 + 1 = 9$

5.) Keep length 4: $p_4 = 9$

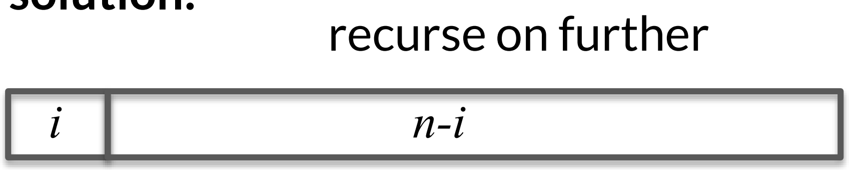
6-8.) Cut into 3 pieces, length 1, 1 and 2 (and all the different orders)

$p_1 + p_1 + p_2 = 7$ $p_1 + p_2 + p_1 = 7$ $p_2 + p_1 + p_1 = 7$

Total: 8 cases for $n = 4$ (= 2^{n-1}). We can slightly reduce by always requiring cuts in non-decreasing order. But still a lot!

Note: We've computed a brute force solution; all possibilities for this simple small example. But we want more optimal solution!

One solution:



recurse on further

i n-i

What are we doing?

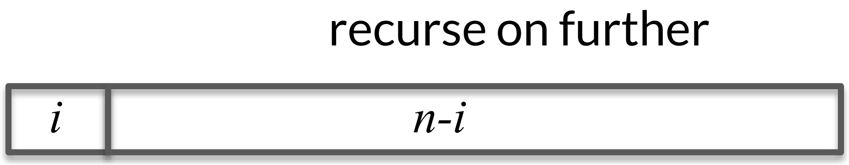
- Cut rod into length i and n-i
- Only remainder n-i can be further cut (recursed)

We need to define:

a.) **Maximum revenue** for log of size *n*: $\boldsymbol{r_n}$ (that is the solution we want to find).

b.) **Revenue (price)** for single log of length *i*: $\boldsymbol{p_i}$

Example: If we cut log into length *i* and *n-i*:



recurse on further

i

n-i

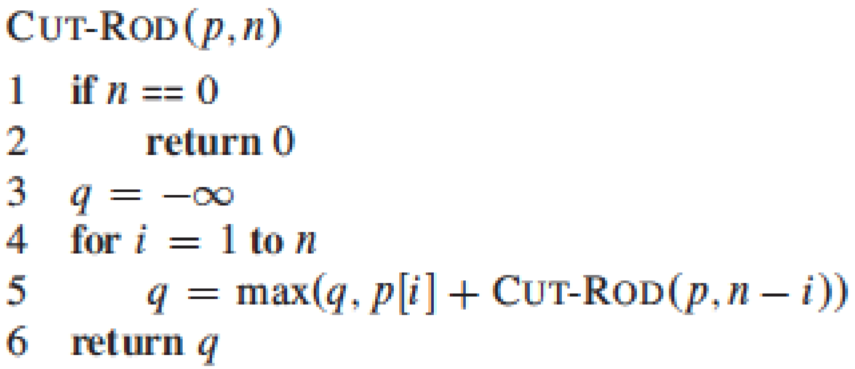
Revenue: $p_i + r_{n-i}$ Can be seen by recursing on *n-i*

What are we going to do?

There are many possible choices of *i*:

$$r_n = \max \left\{ \begin{matrix} p_1 + r_{n\text{-}1} \\ p_1 + r_{n\text{-}2} \\ \ldots \\ p_n + r_0 \end{matrix} \right\}$$

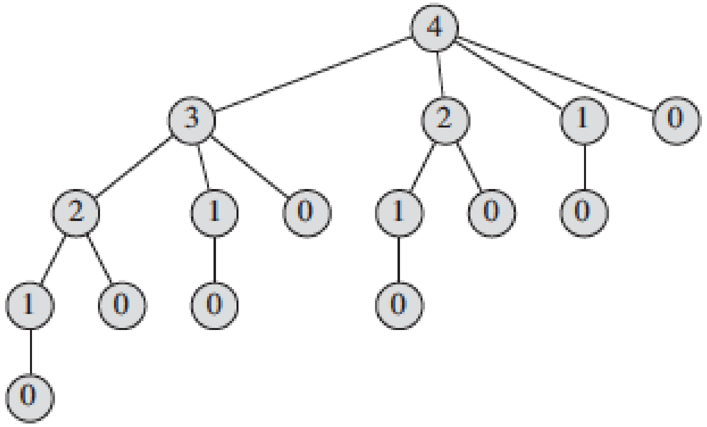
Recursive (top-down) pseudo code:



```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

Problem? Slow runtime (it’s essential brute force)!

But why? Cut-rod calls itself repeatedly with the same parameter values (tree):



- **Node label:** size of the subproblem called on
- Can be seen by eye that many subproblems are called repeatedly (subproblem overlap)
- Number of nodes exponential in n (2^n). therefore exponential number of calls.

- Therefore … **Dynamic Programing Approach:**

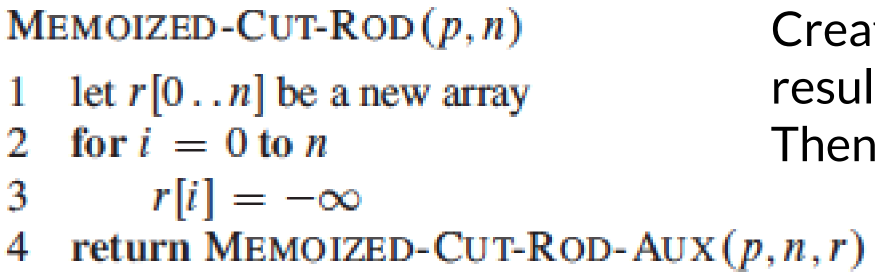
- Recursive solution is inefficient, since it repeatedly calculates a solution of the same subproblem (overlapping subproblem).

- Instead, solve each subproblem only once AND save its solution. Next time we encounter the subproblem look it up in a hashtable or an array (**Memoization**, recursive top-down solution).

- We will also talk about a second solution where we save the solution of subproblems Of increasing size (i.e. in order) in an array. Each time we will fall back on solutions that we obtained in previous steps and stored in an array (bottom-up solution).

Recursive top-down solution: Cut-Rod with Memoization

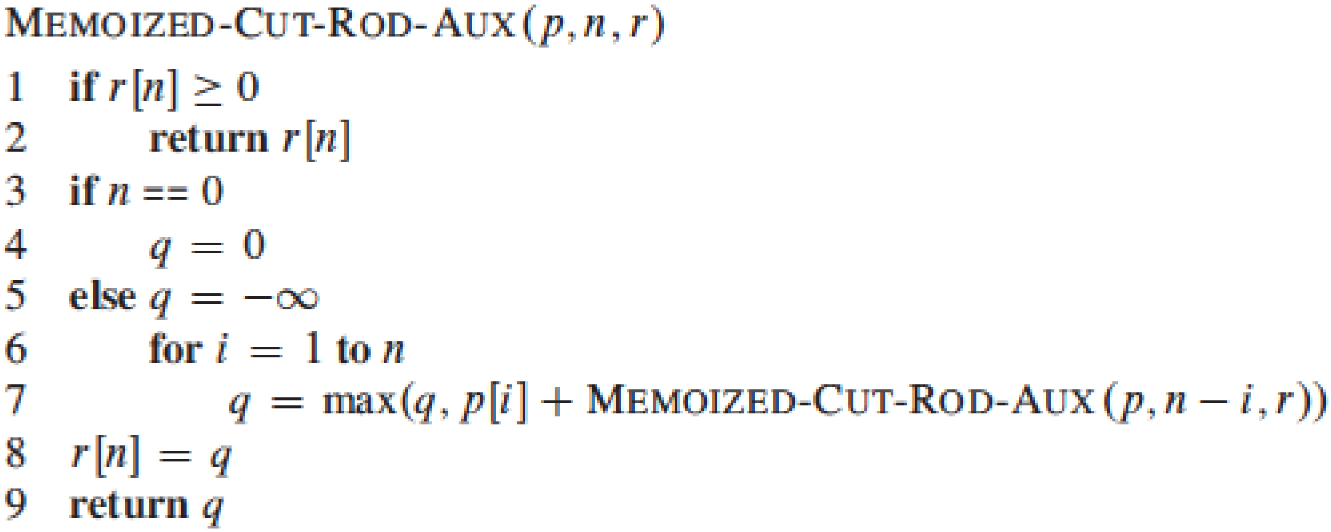
Step 1 Initialization:



```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

Creates array for holding memoized results, and initialized to minus infinity. Then calls the main auxiliary function.

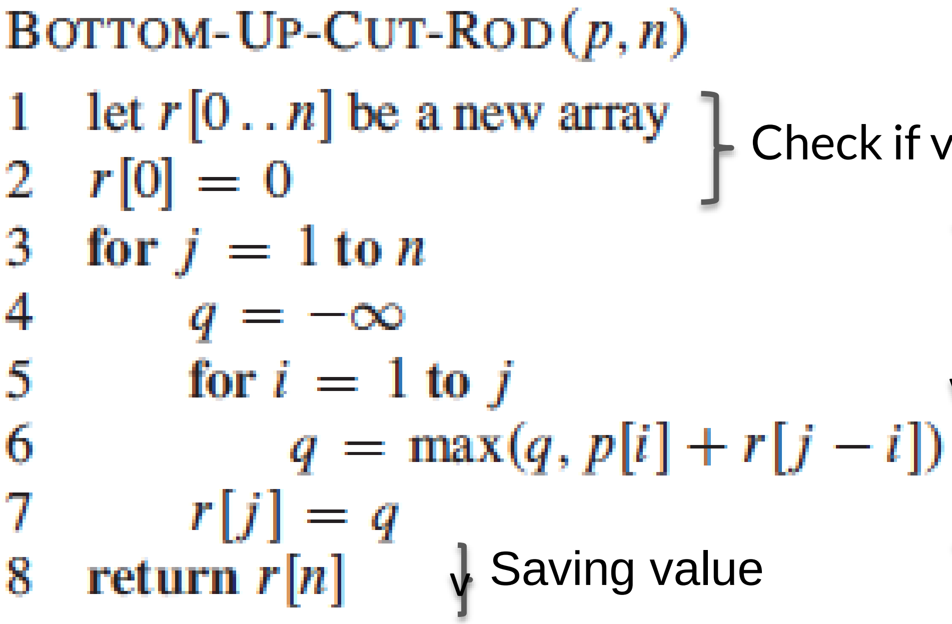
Step 2: The main auxiliary function, which goes through the lengths, computes answers to subproblems and memoizes if subproblem not yet encountered:



```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
        return r[n]
if n == 0
        q = 0
else q = −∞
        for i = 1 to n
                q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Bottom-up solution: Bottom Up Cut-Rod

Each time we use previous values form arrays:



```
BOTTOM-UP-CUT-ROD($p, n$)
let $r[0 \ldots n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

Check if value already known or memoized

Compute maximum revenue if it hasn't already been computed.

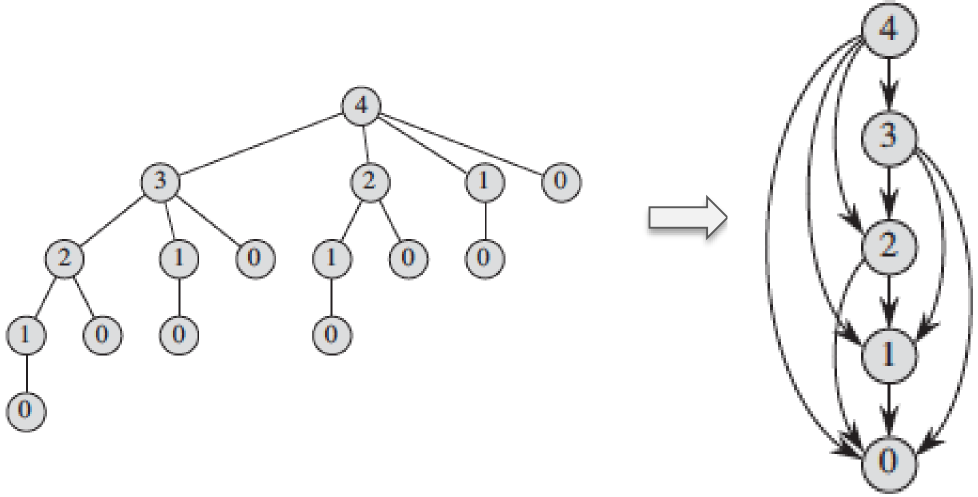
Saving value

Run time: For bottom-up and top-down approach $O(n^2)$

Why (double nested loop)?

We can also view the subproblems encountered in graph form.

- We reduce the previous tree that included all the subproblems repeatedly



- each vertex represents subproblem of a given size
- Vertex label: subproblem size
- Edges from *x* to *y*: We need a solution for subproblem *x* when solving subproblem *y*

Run time: Can be seen as number of edges $O(n^2)$

Note: Run time is a combination of number of items in table *(n)* and work per item *(n)*.

The work per item because of the max operation (needed even if the table is filled.

And we just take values from the table) is proportional to *n* as in the number of edges in graph.

Greedy Algorithms

Optimization Problems

- Shortest path is an example of an optimization problem: we wish to find the path with lowest weight.
- What is the general character of an optimization problem?

Optimization Problems

- Ingredients:

  Instances: The possible inputs to the problem.

  Solutions for Instance: Each instance has an exponentially large set of valid solutions.

  Cost of Solution: Each solution has an easy-to-compute cost or value.

- Specification

  Preconditions: The input is one instance.

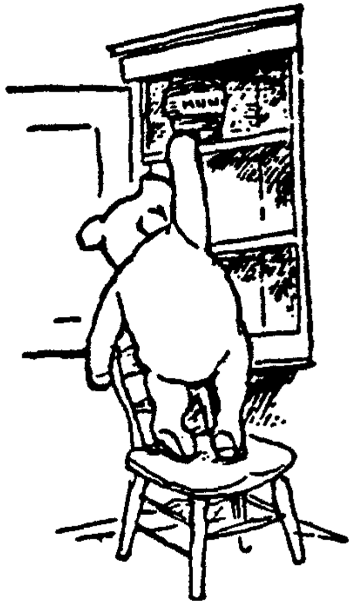
  Postconditions: A valid solution with optimal cost. (minimum or maximum)

Greedy Solutions to Optimization Problems

Every two-year-old knows the greedy algorithm.

In order to get what you want, just start grabbing what looks best.

Surprisingly, many important and practical optimization problems can be solved this way.



Example 1: Making Change

- **Problem:** Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

The Greedy Choice

Commit to the object that looks the ``best''

Must prove that this locally greedy choice does not have negative global consequences.

Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢

25¢	25¢	[✓ 25¢]	25¢	25¢	[✓ 25¢]	25¢	25¢	25¢	25¢
10¢	10¢	10¢	[✓ 10¢]	10¢	10¢	10¢	10¢	10¢	[✓ 10¢]
5¢	[✓ 5¢]	5¢	5¢	[✓ 5¢]	5¢	5¢	[✓ 5¢]	5¢	5¢
[✓ 1¢]	1¢	[✓ 1¢]	[✓ 1¢]	[✓ 1¢]	1¢	[✓ 1¢]	[✓ 1¢]	[✓ 1¢]	1¢

Solutions for Instance: A subset of the coins in the drawer that total the amount

Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢

25¢	25¢	[✓ 25¢]	25¢	25¢	[✓ 25¢]	25¢	25¢	25¢	25¢
10¢	10¢	10¢	[✓ 10¢]	10¢	10¢	10¢	10¢	10¢	[✓ 10¢]
5¢	[✓ 5¢]	5¢	5¢	[✓ 5¢]	5¢	5¢	[✓ 5¢]	5¢	5¢
[✓ 1¢]	1¢	[✓ 1¢]	[✓ 1¢]	[✓ 1¢]	1¢	[✓ 1¢]	[✓ 1¢]	[✓ 1¢]	1¢

Solutions for Instance: A subset of the coins that total the amount.

Cost of Solution: The number of coins in the solution = 14

Goal: Find an optimal valid solution.

Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢

25¢	25¢	[✓ 25¢]	25¢	25¢	[✓ 25¢]	25¢	[✓ 25¢]	25¢	25¢
10¢	10¢	10¢	[✓ 10¢]	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	[✓ 5¢]	5¢	5¢	5¢	5¢	5¢
1¢	1¢	[✓ 1¢]	1¢	[✓ 1¢]	1¢	1¢	1¢	1¢	1¢

Greedy Choice:

Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Cost of Solution: 7

Hard Making Change Example

- **Problem:** Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4-cent coin.

Consequences:

4+1+1 = 6 mistake

3+3=6 better

Greedy Algorithm does not work!

When Does It Work?

Greedy Algorithms: Easy to understand and to code, but do they work?

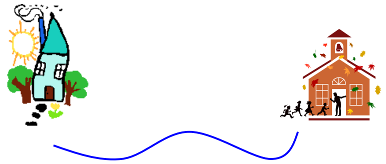
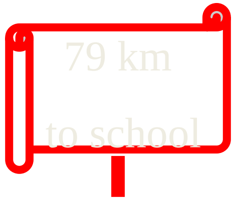
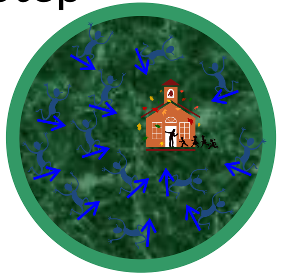
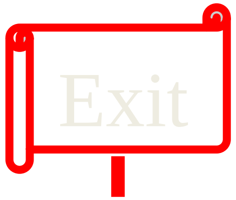
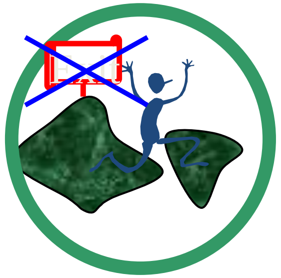
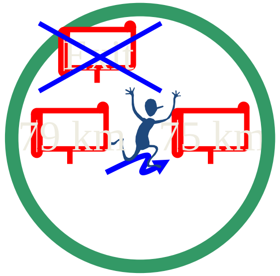
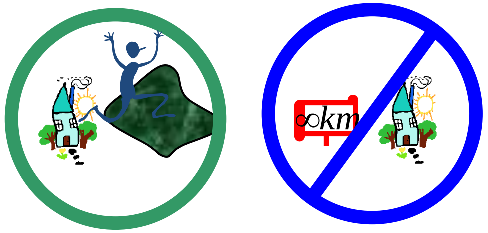
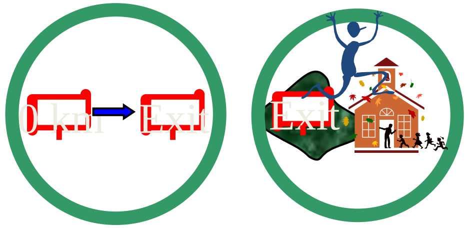
For most optimization problems, all greedy algorithms tried do not work (i.e. yield sub-optimal solutions)

But some problems can be solved optimally by a greedy algorithm.

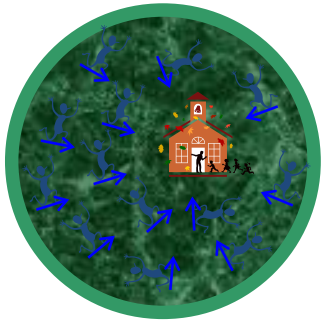
The proof that they work, however, is subtle.

As with all iterative algorithms, we use loop invariants.

Designing an Algorithm

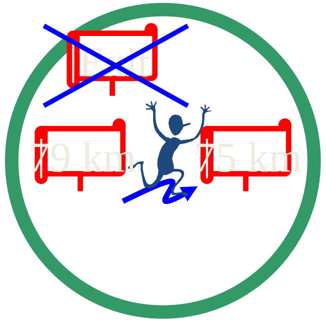
Define Problem 	Define Loop Invariants 	Define Measure of Progress 
Define Step 	Define Exit Condition 	Maintain Loop Inv 
Make Progress 	Initial Conditions 	Ending 

Define Step



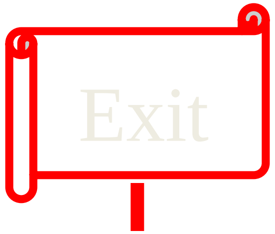
The algorithm chooses the “best” object from amongst those not considered so far and either commits to it or rejects it.

Make Progress



Another object considered

Exit Condition

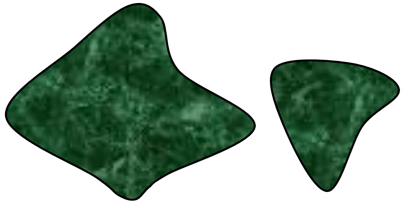


All objects have been considered

Designing a Greedy Algorithm

```
< pre-condition >
CodeA
loop
< loop-invariant>
while ¬ exit condition
    CodeB
end loop
CodeC
< post-condition >
```

Loop Invariant



We have not gone wrong.
There is at least one optimal solution consistent with the choices made so far.

Establishing the Loop Invariant

Establishing Loop Invariant

<preCond>
codeA → <loop-invariant>

Initially no choices have been made and hence all optimal solutions are consistent with these choices.

Maintaining Loop Invariant

Must show that $<$loop-invariant$>$ + CodeB $\rightarrow$ $<$loop-invariant$>$

$<$LI$>$: $\exists$ optimal solution $OptS_{LI}$ consistent with choices so far

CodeB : Commit to or reject next object

$<$LI$>$: $\exists$ optimal soln $OptS_{Ours}$ consistent with prev objects + new object

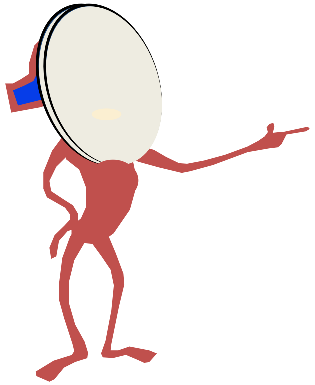
Note : $OptS_{Ours}$ may or may not be the same as $OptS_{LI}$!

Proof must massage $optS_{LI}$ into $optS_{ours}$ and prove that $optS_{ours}$:

- is a valid solution
- is consistent both with previous and new choices.
- is optimal

Three Players

Algorithm:
commits to or rejects next best object



Prover:
Proves LI is maintained.

His actions are not part of the algorithm

Fairy God Mother:
Holds the hypothetical optimal sol optSLI.

The algorithm and prover do not know optSLI.

Proving the Loop Invariant is Maintained

We need to show that the action taken by the algorithm maintains the loop invariant.

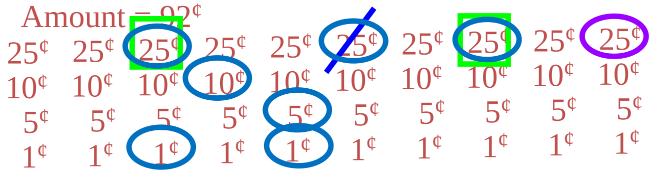
There are 2 possible actions:

Case 1. Commit to current object

Case 2. Reject current object

Case 1 :
Committing to Current Object

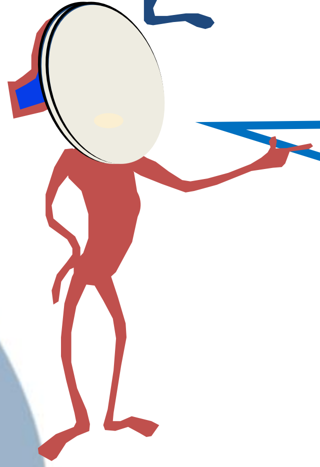
Massaging optSLI into optSours



optSLI

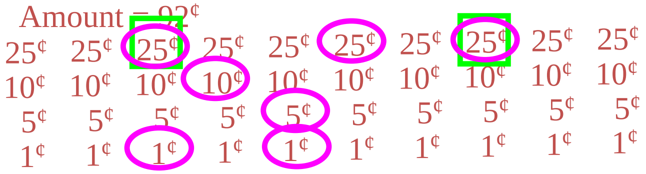
I hold $\text{optS}_{\text{ours}}$ witnessing that there is an opt sol consistent with previous & new choices.

I commit to keeping another 25¢



I instruct how to massage optSLI into optSours so that it is consistent with previous & new choice.

As Time Goes On



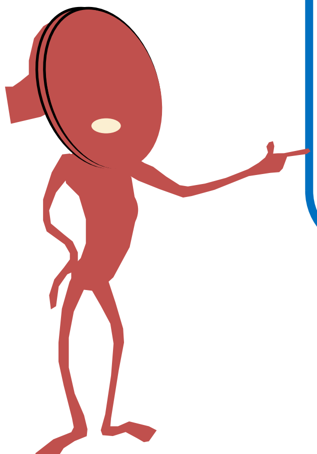
Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS_{LI}

I always hold an opt sol optSLI but one that keeps changing.

I keep making more choices.

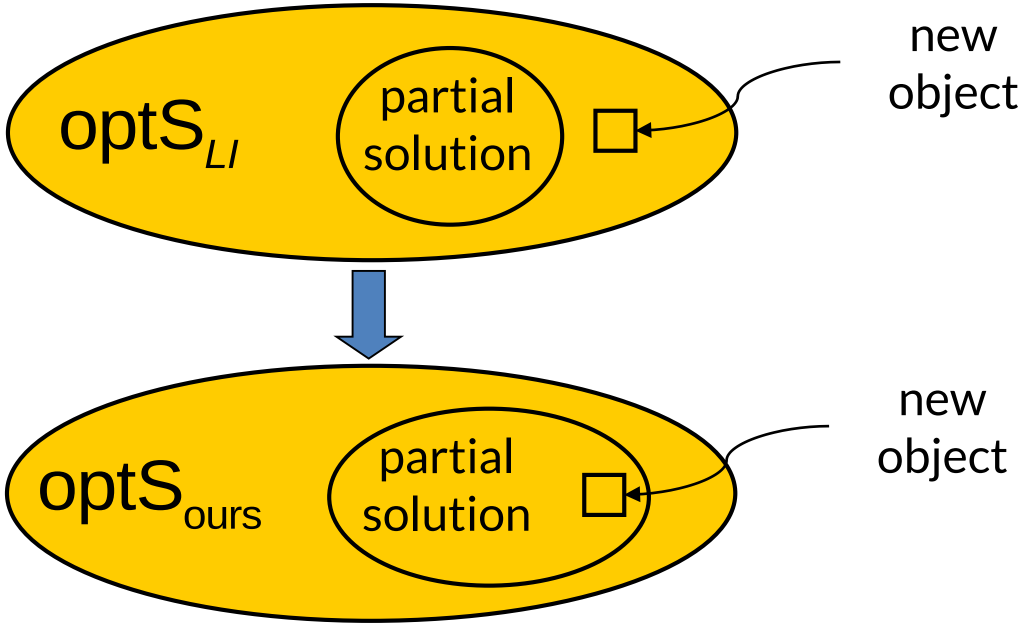


I know that her optSLI
is consistent with these choices.

Hence, I know more and more of optSLI

In the end, I know it all.

Case 1A:
The object we commit to is already part of optSLI

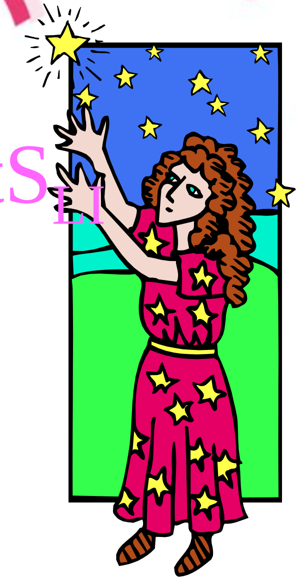


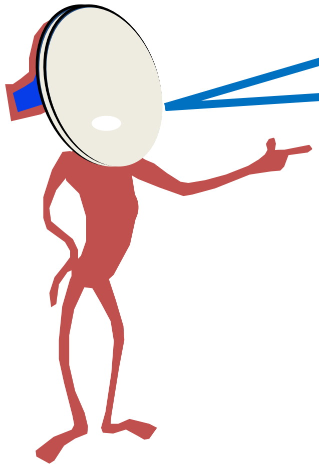
Massaging optSLI into optSours

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

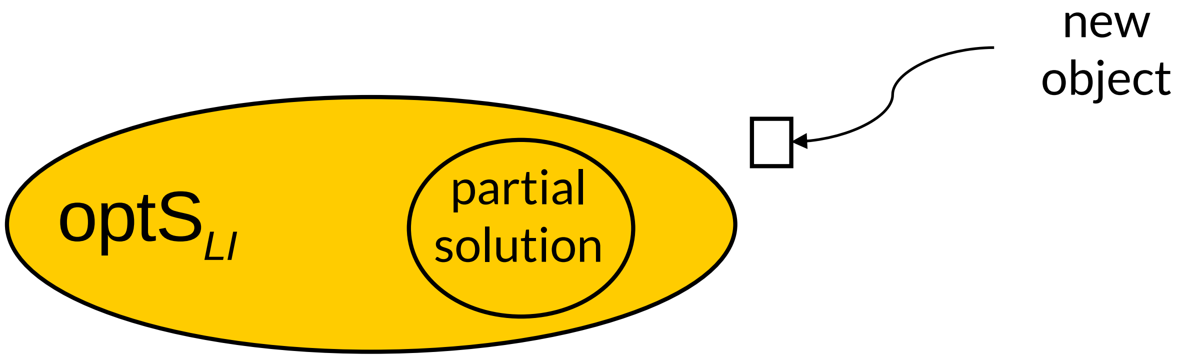
optS_{LI}





If it happens to be the case that the new object selected is consistent with the solution held by the fairy godmother, then we are done.

Case 1B:
The object we commit to is not part of optSLI



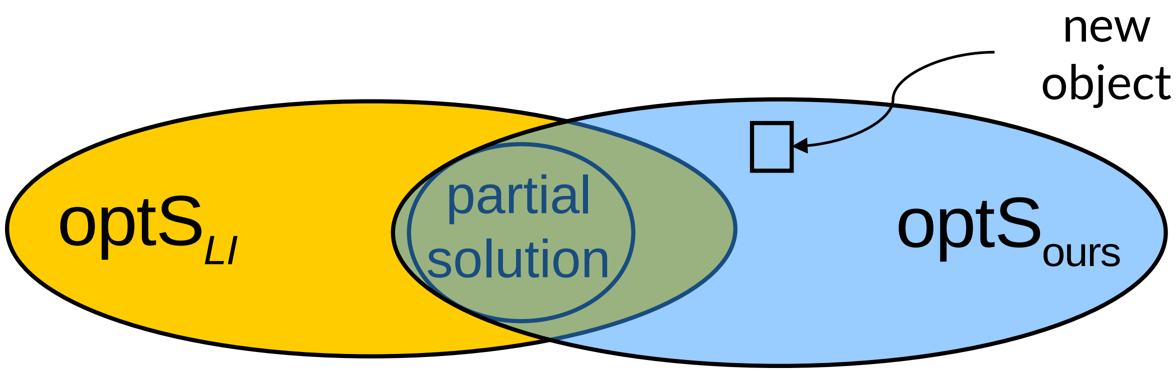
Case 1B:
The object we commit to is not part of optSLI

This means that our partial solution is not consistent with optS$_{LI}$.

The Prover must show that there is a new optimal solution optS$_{ours}$ that is consistent with our partial solution.

This has two parts

- All objects previously committed to must be part of optS$_{ours}$.
- The new object must be part of optS$_{ours}$.

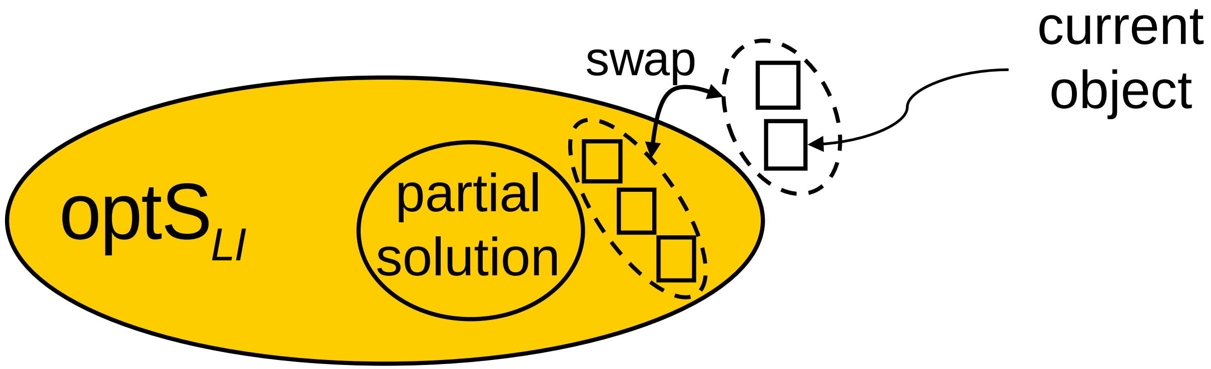


Case 1B:
The object we commit to is not part of optSLI

Strategy of proof: construct a consistent optS$_{ours}$ by replacing one or more objects in optS$_{LI}$ (but not in the partial solution) with another set of objects that includes the current object.

We must show that the resulting optS$_{ours}$ is still

- Valid
- Consistent
- Optimal

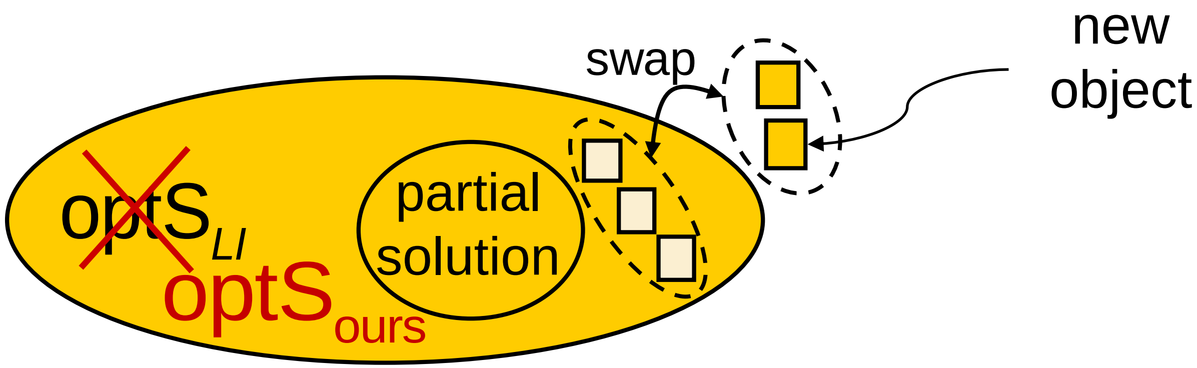


Case 1B:
The object we commit to is not part of optSLI

Strategy of proof: construct a consistent optS$_{ours}$ by replacing one or more objects in optS$_{LI}$ (but not in the partial solution) with another set of objects that includes the current object.

We must show that the resulting optS$_{ours}$ is still

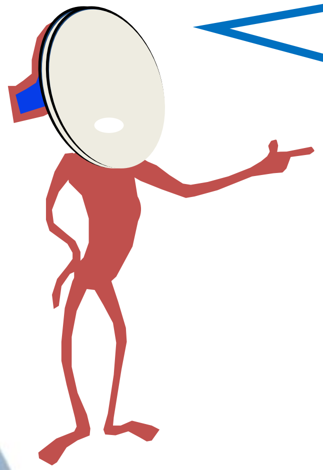
- Valid
- Consistent
- Optimal



Massaging optSLI into optSours

Amount = 92¢

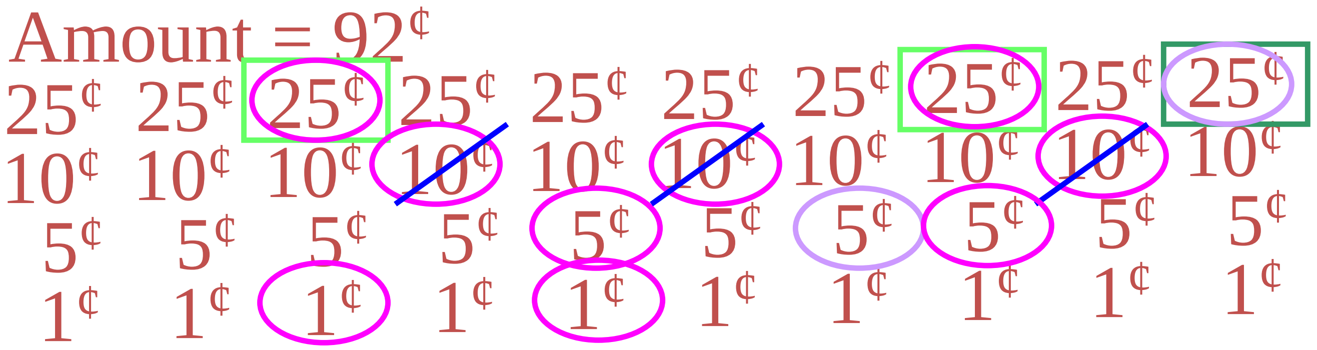
25¢	25¢	25¢	25¢	25¢	~~25¢~~	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢



Replace	With
A different 25¢	Alg's 25¢

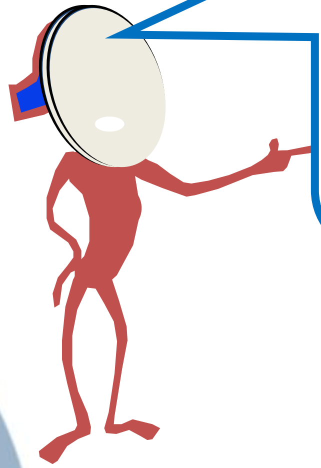
Massaging optSLI into optSours

Amount = 92¢



25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	~~10¢~~	10¢	~~10¢~~	10¢	10¢	~~10¢~~	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS$_{LI}$



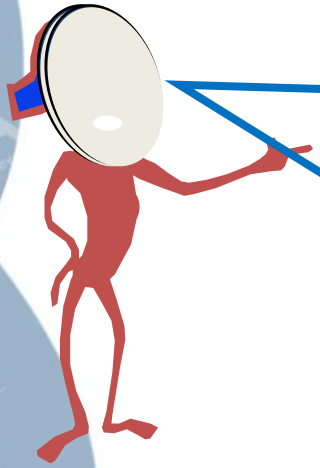
Replace	With
A different 25¢	Alg’s 25¢
3×10¢	Alg’s 25¢ + 5¢

Massaging optSLI into optSours

Amount = 92¢

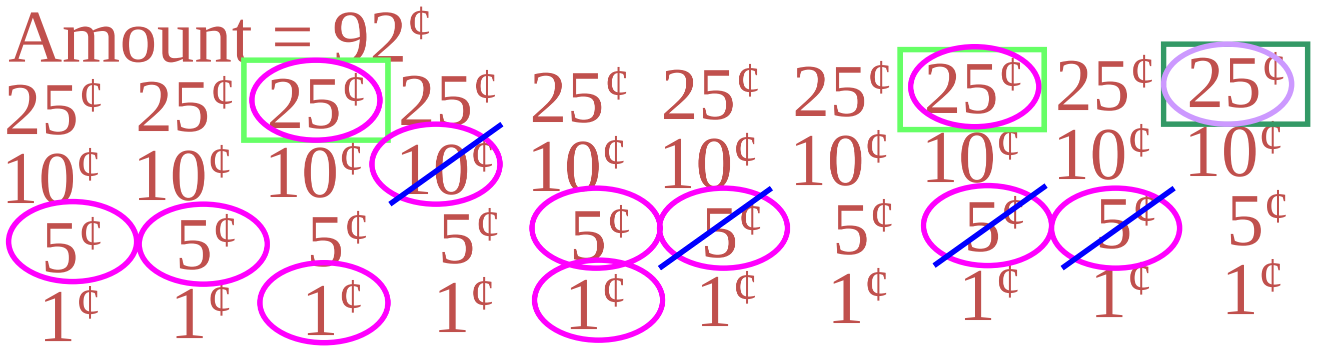
25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	~~10¢~~	10¢	~~10¢~~	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	~~5¢~~	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS_{LI}



Replace	With
A different 25¢	Alg's 25¢
3×10¢	Alg's 25¢ + 5¢
2×10¢ + 1×5¢	Alg's 25¢

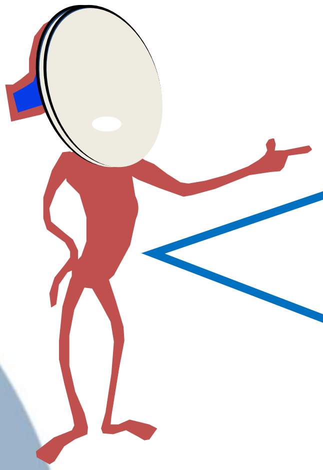
Massaging optSLI into optSours



Amount = 92¢

25¢	25¢	$\boxed{25¢}$	25¢	25¢	25¢	25¢	$\boxed{25¢}$	25¢	$\boxed{25¢}$
10¢	10¢	10¢	$\cancel{10¢}$	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	$\cancel{5¢}$	5¢	$\cancel{5¢}$	$\cancel{5¢}$	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS_{LI}

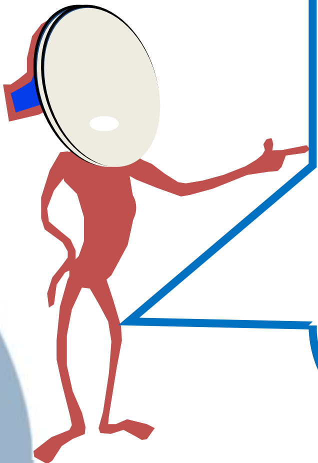


Replace	With
A different 25¢	Alg’s 25¢
3×10¢	Alg’s 25¢ + 5¢
2×10¢ + 1×5¢	Alg’s 25¢
1×10¢ + 3×5¢	Alg’s 25¢

Massaging optSLI into optSours

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	~~10¢~~	10¢	10¢	~~10¢~~	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	~~1¢~~	~~1¢~~	~~1¢~~	~~1¢~~	~~1¢~~



Replace	With
A different 25¢	Alg's 25¢
3×10¢	Alg's 25¢ + 5¢
2×10¢ + 1×5¢	Alg's 25¢
1×10¢ + 3×5¢	Alg's 25¢
?? + 5×1¢	Alg's 25¢

Must Consider All Cases

	#Coins		#Coins
1Q	1	1Q	1
3D	3	1Q 1N	2
2D 1N	3	1Q	1
2D 5P	7	1Q	1
1D 3N	4	1Q	1
1D 2N 5P	8	1Q	1
1D 1N 10P	12	1Q	1
1D 15P	16	1Q	1
5N	5	1Q	1
4N 5P	9	1Q	1
3N 10P	13	1Q	1
2N 15P	17	1Q	1
1N 20P	21	1Q	1
25P	25	1Q	1

Note that in all cases our new solution $optS_{ours}$ is:

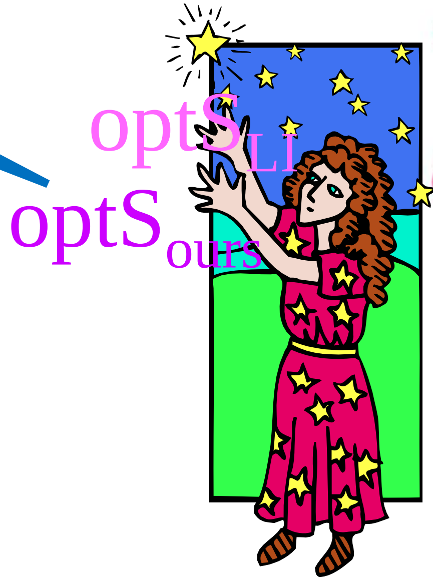
Valid: the sum is still correct

Consistent with our previous choices (we do not alter these).

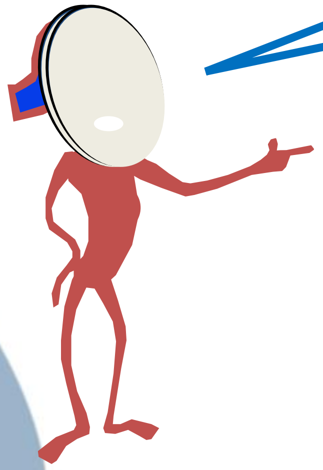
Optimal: we never add more coins to the solution than we delete

Massaging optSLI into optSours

Done



She now has something.
We must prove that it is what we want.



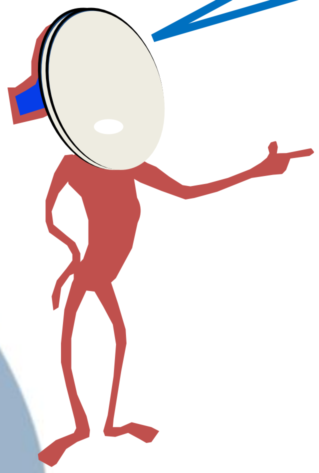
Massaging optSLI into optSours

$\text{optS}_{\text{ours}}$

$\text{optS}_{\text{ours}}$ is valid

optS_{LI} was valid and we introduced no new conflicts.

Total remains unchanged.

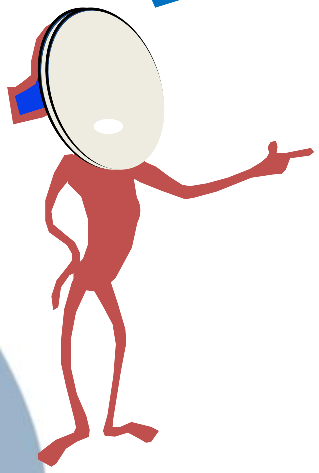


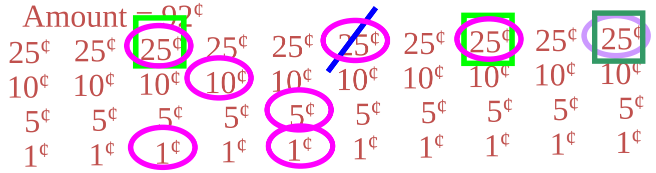
Replace	With
A different 25¢	Alg’s 25¢
3×10¢	Alg’s 25¢ + 5¢
2×10¢ + 1×5¢	Alg’s 25¢
1×10¢ + 3×5¢	Alg’s 25¢
?? + 5×1¢	

Massaging optSLI into optSours

$\text{optS}_{\text{ours}}$ is consistent

optSLI was consistent with previous choices and we made it consistent with new.





Amount = 92¢

25¢	25¢	25¢	25¢	25¢	~~25¢~~	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

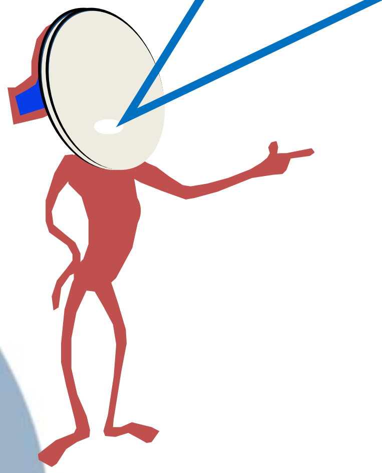
Massaging optSLI into optSours

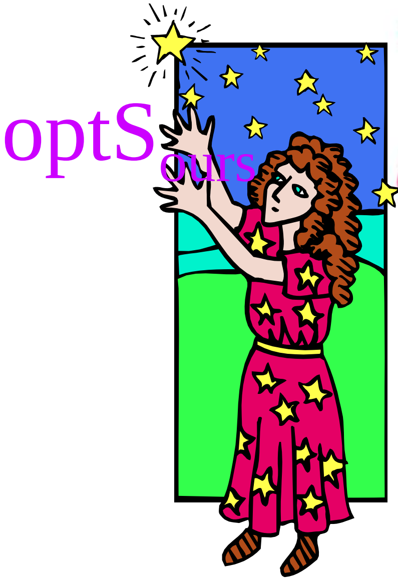
$\text{optS}_{\text{ours}}$ is optimal

We do not even know the cost of an optimal solution.

optS_{LI} was optimal and

$\text{optS}_{\text{ours}}$ cost (# of coins) is not bigger.





Replace	With
A different 25¢	Alg’s 25¢
3×10¢	Alg’s 25¢ + 5¢
2×10¢ + 1×5¢	Alg’s 25¢
1×10¢ + 3×5¢	Alg’s 25¢
?? + 5×1¢	

Committing to Other Coins

Similarly, we must show that when the algorithm selects a dime, nickel or penny, there is still an optimal solution consistent with this choice.

$$optS_{LI} \xrightarrow{+\text{dime}} optS_{Ours}$$

$$optS_{LI} \xrightarrow{+\text{nickel}} optS_{Ours}$$

$$optS_{LI} \xrightarrow{+\text{penny}} optS_{Ours}$$

Example: Dimes

We only commit to a dime when less than 25¢ is unaccounted for.

Therefore the coins in optSLI that this dime replaces have to be dimes, nickels or pennies.

optS_{LI}	#Coins		$\text{optS}_{\text{Ours}}$	#Coins
1D	1		1D	1
2N	2		1D	1
1N 5P	6		1D	1
10P	10		1D	1

Committing to Other Coins

We must consider all possible coins we might select:

- **Quarter:** Swap for another quarter, 3 dimes (with a nickel) etc.
- **Dime:** Swap for another dime, 2 nickels, 1 nickel + 5 pennies etc.
- **Nickel:** Swap for another nickel or 5 pennies.
- **Penny:** Swap for another penny.

Massaging optSLI into optSours

Case 1

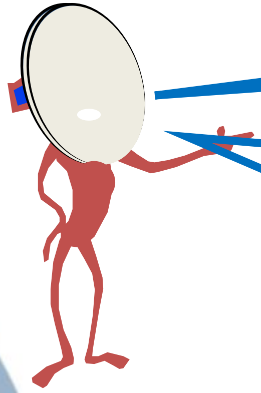
$\text{optS}_{\text{ours}}$ is valid

$\text{optS}_{\text{ours}}$ is consistent

$\text{optS}_{\text{ours}}$ is optimal

$\text{optS}_{\text{ours}}$

$\text{optS}_{\text{ours}}$ → <LI>



Maintaining Loop Invariant

<LI>
¬<exit Cond>
codeB
→ <LI>

Case 2:
Rejecting the Current Object

Rejecting the Current Object

Strategy of Proof:

1. There is at least one optimal solution $\mathrm{optS}_{\mathrm{LI}}$ consistent with previous choices.

2. Any optimal solution consistent with previous choices cannot include current object.

3. Therefore $\mathrm{optS}_{\mathrm{LI}}$ cannot include current object.

Rejecting an Object

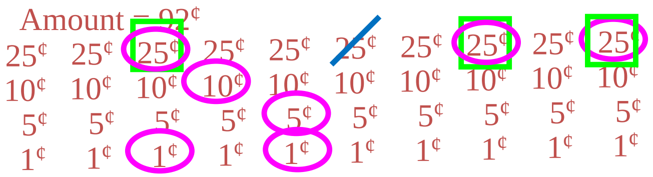
Making Change Example:

We only reject an object when including it would make us exceed the total.

Thus optS_{LI} cannot include the object either.

Massaging optSLI into optSours

Case 2

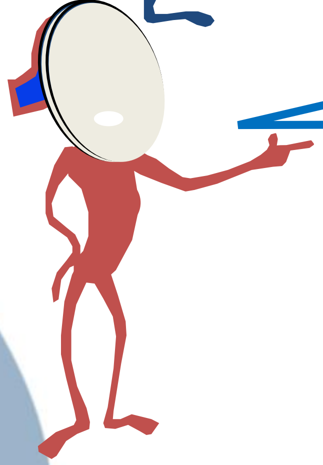


Amount = 92¢

25¢	25¢	25¢	25¢	25¢	~~25¢~~	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

I hold optSLI witnessing that there is an opt sol consistent with previous choices.

I reject the next 25¢



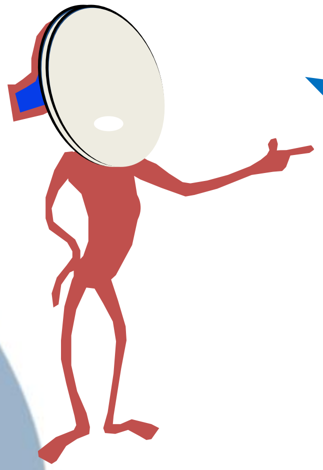
I must make sure that what the Fairy God Mother has is consistent with this new choice.

Massaging optSLI into optSours

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	~~25¢~~	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

$optS_{LI}$



The Algorithm has
92¢-75¢ = 17¢ < 25¢ unchoosen.

Fairy God Mother must
have < 25¢ that I don’t know about.

$optS_{LI}$ does not contain the 25¢ either.

Clean up loose ends

<loop-invariant>
<exit Cond>
codeC
→ <postCond>

<exit Cond> → Alg has committed to or rejected each object.

Has yielded a solution S.

<LI> → $ opt sol consistent with these choices.

S must be optimal.

codeC → Alg returns S .

→ <postCond>

Making Change Example

Problem: Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

Greedy Choice: Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Yes

Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.

Massaging optSLI into optSours

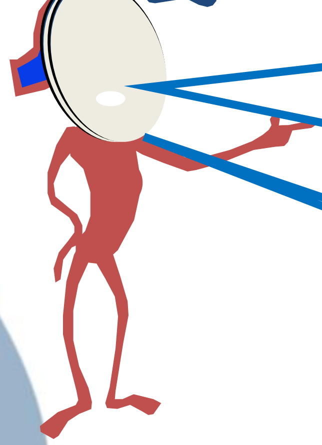
Amount = 6¢

4¢	4¢	4¢	4¢	4¢	4¢	4¢	4¢	4¢	4¢
3¢	3¢	3¢	3¢	3¢	3¢	3¢	3¢	3¢	3¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS_{LI}

I hold optS_{LI}.

I commit to keeping a 4¢



I will now instruct how to massage optS_{LI} into $\text{optS}_{\text{ours}}$ so that it is consistent with previous & new choice.

Oops!

Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.

Consequences:

4+1+1 = 6   mistake

3+3=6   better

Greedy Algorithm does not work!

Analysing Arbitrary Systems of Denominations

Suppose we are given a system of coin denominations. How do we decide whether the greedy algorithm is optimal?

It turns out that this problem can be solved in $O(D^3)$ time, where D = number of denominations (e.g., $D=6$ in Canada) (Pearson 1994).

Designing Optimal Systems of Denominations

In Canada, we use a 6 coin system:
1 cent, 5 cents, 10 cents, 25 cents, 100 cents and 200 cents.

Assuming that N, the change to be made, is uniformly distributed over $\{1,\ldots,499\}$, the expected number of coins per transaction is 5.9.

The optimal (but non-greedy) 6-coin systems are $(1,6,14,62,99,140)$ and $(1,8,13,69,110,160)$, each of which give an expected 4.67 coins per transaction.

The optimal *greedy* 6-coin systems are (1,3,8,26,64,{202 or 203 or 204}) and (1,3,10,25,79,{195 or 196 or 197}) with an expected cost of 5.036 coins per transaction.

Summary

We must prove that every coin chosen or rejected in greedy fashion still leaves us with a solution that is

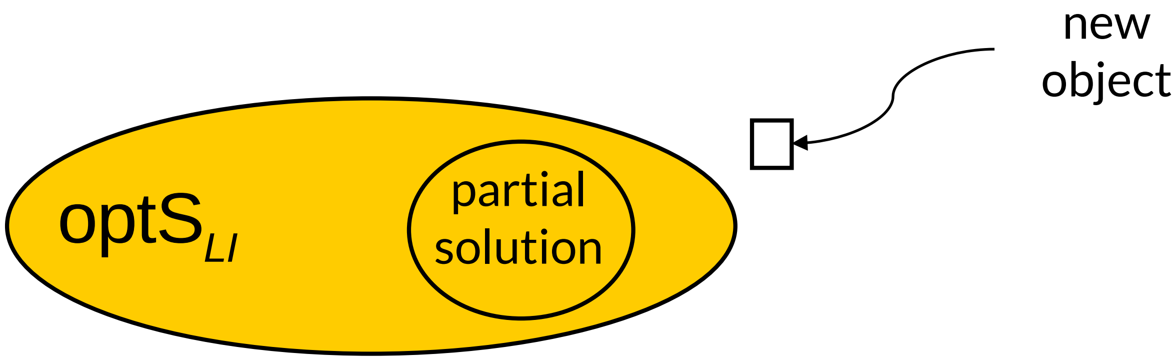
Valid

Consistent

Optimal

We prove this using an inductive ‘cut and paste’ method.

We know from the previous iteration we have a partial solution ***S_{part}*** that is part of some complete optimal solution ***$optS_{LI}$***.



Summary

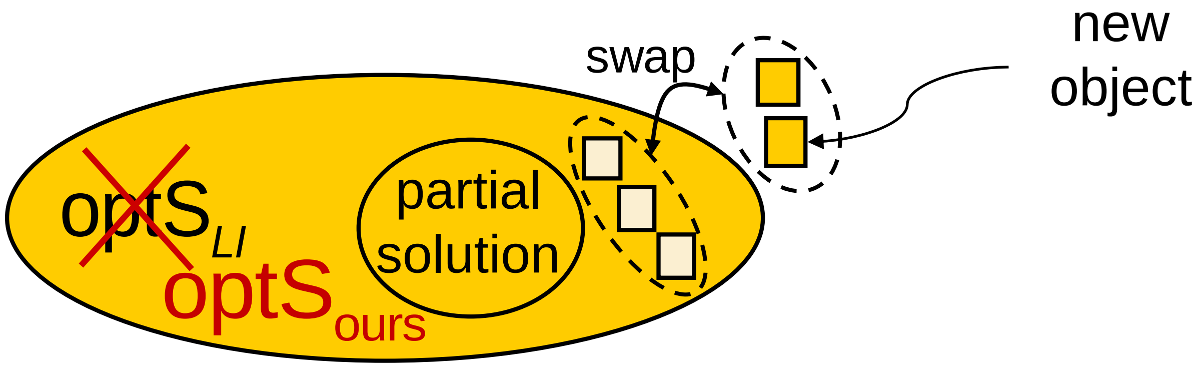
Selecting a coin: we show that we can replace a subset of the coins in ***optS*$_{LI}$ \ *S*$_{part}$** with the selected coin (+ perhaps some additional coins).

Valid because we ensure that the trade is fair (sums are equal)

Consistent because we have not touched ***S*$_{part}$**

Optimal because the number of the new coin(s) is no greater than the number of coins they replace.

Rejecting a coin: we show that we only reject a coin when it could not be part of ***optS*$_{LI}$**.



Example 2: Job/Event Scheduling

The Job/Event Scheduling Problem

- **Ingredients:**

Instances: Events with starting and finishing times

$<<s_1,f_1>,<s_2,f_2>,\ldots,<s_n,f_n>>$.

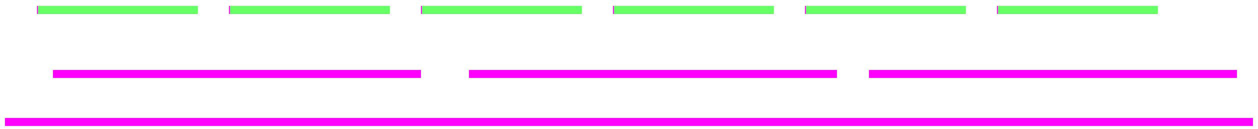
Solutions: A set of events that do not overlap.

Value of Solution: The number of events scheduled.

Goal: Given a set of events, schedule as many as possible.

Example: Scheduling lectures in a lecture hall.

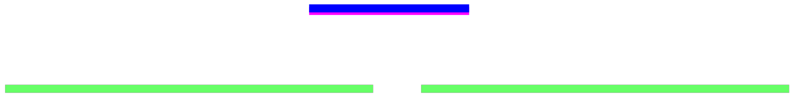
Possible Criteria for Defining “Best”



Optimal

Greedy Criterion: The Shortest Event

Motivation: Does not book the room for a long period of time.

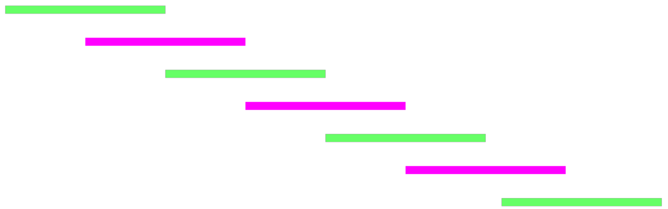


Schedule first

Optimal

Counter Example

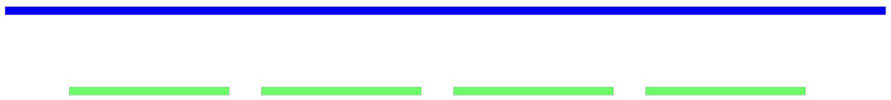
Possible Criteria for Defining “Best”



Optimal

Greedy Criterion: The Earliest Starting Time

Motivation: Gets room in use as early as possible

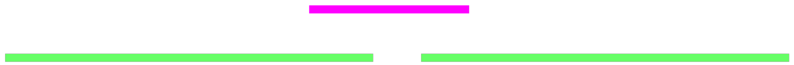


Schedule first

Optimal

Counter Example

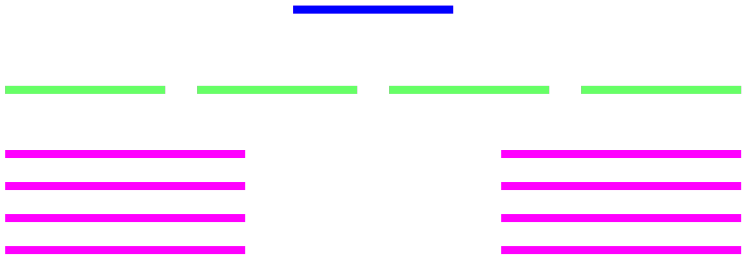
Possible Criteria for Defining "Best"



Optimal

Greedy Criterion: Conflicting with the Fewest Other Events

Motivation: Leaves many that can still be scheduled.

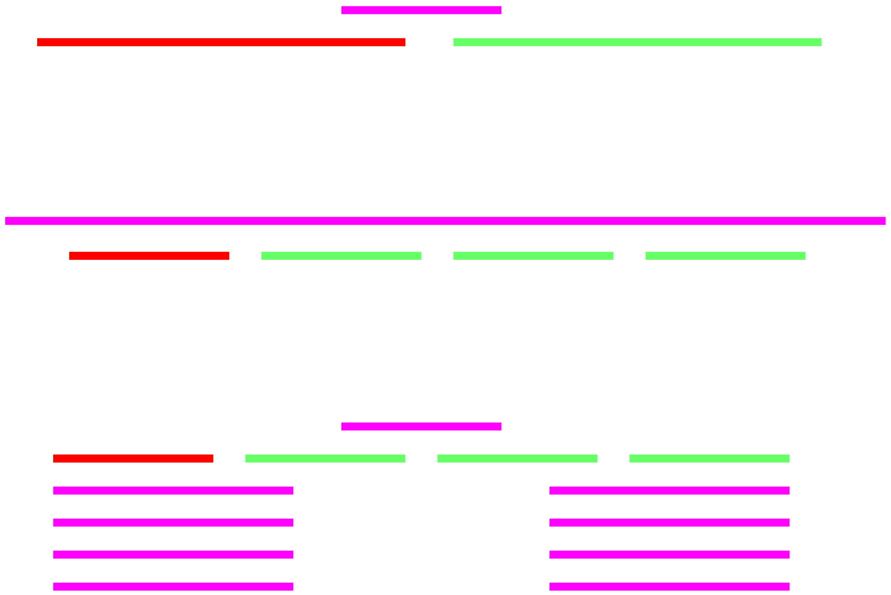


Schedule first

Optimal

Counter Example

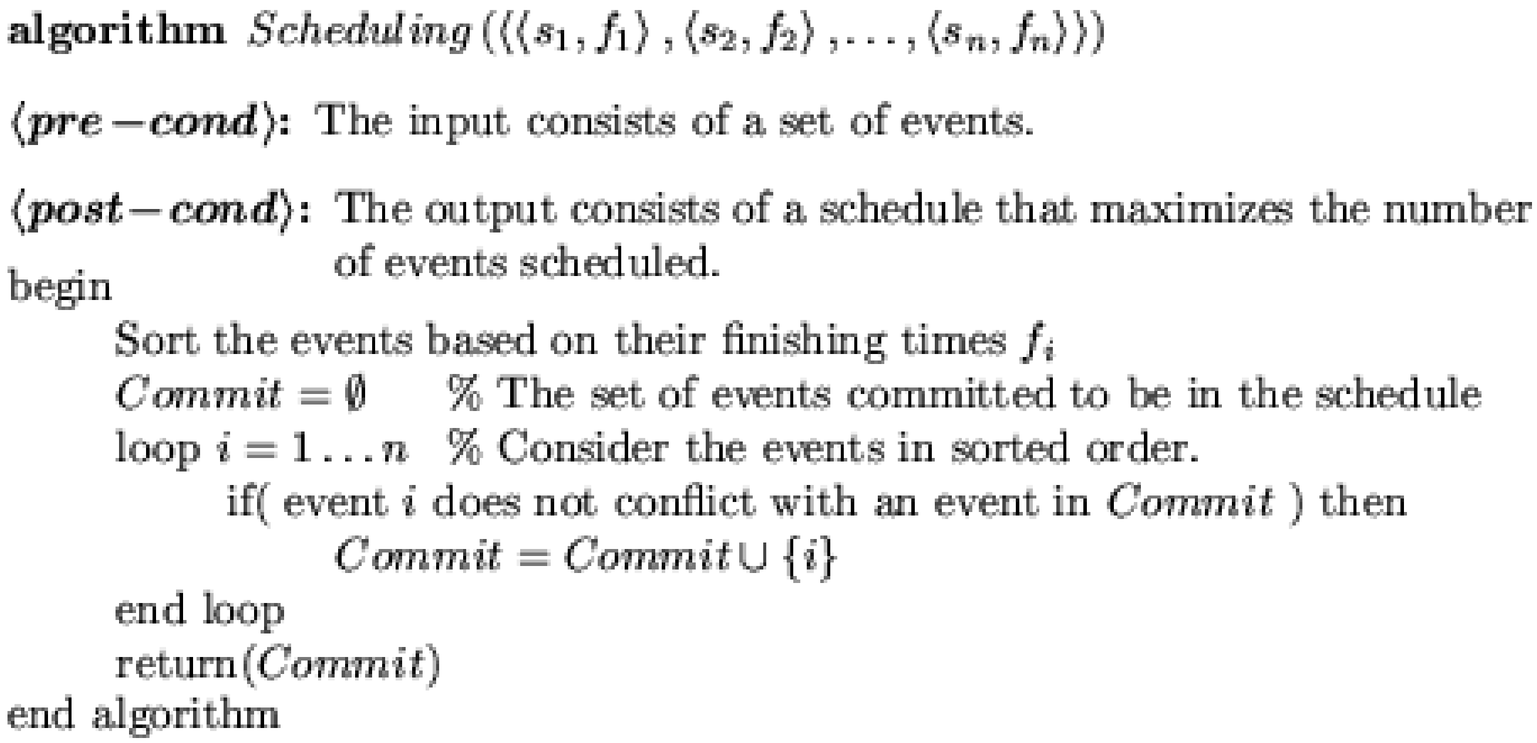
Possible Criteria for Defining "Best"



Greedy Criterion: Earliest Finishing Time

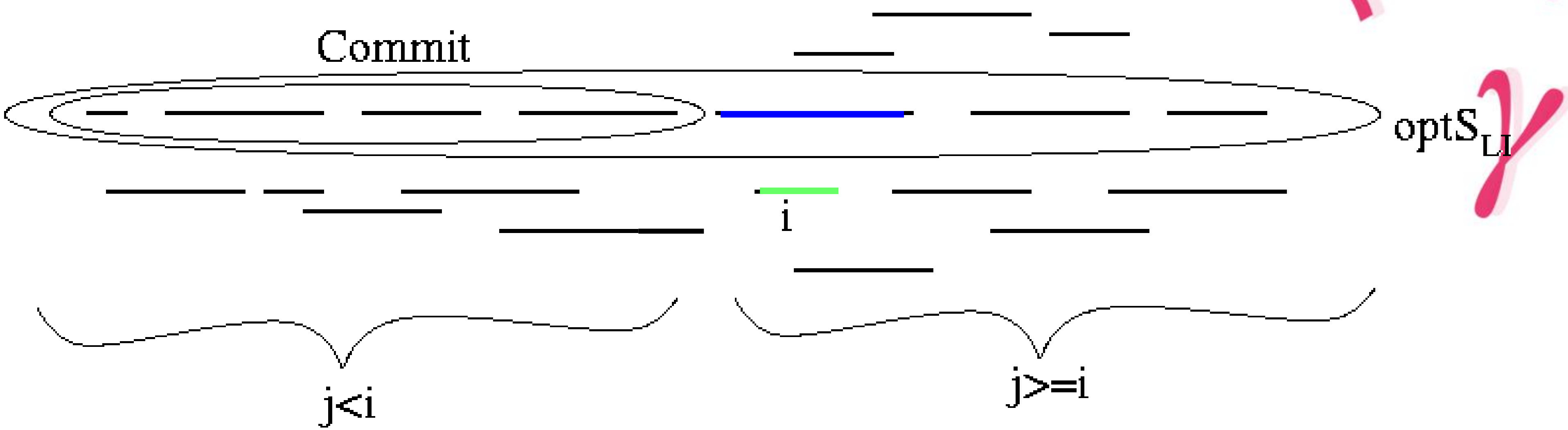
Motivation: Schedule the event that will free up your room for someone else as soon as possible.

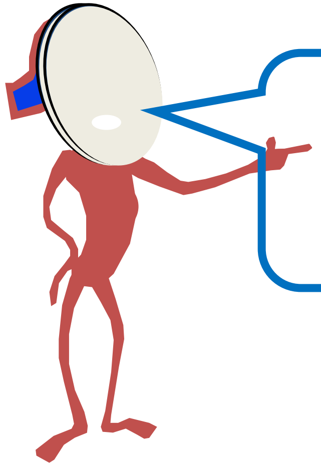
The Greedy Algorithm



```
**algorithm** $Scheduling(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle)$

$\langle pre-cond\rangle$: The input consists of a set of events.

$\langle post-cond\rangle$: The output consists of a schedule that maximizes the number
                 of events scheduled.
begin
    Sort the events based on their finishing times $f_i$
    $Commit = \emptyset$    % The set of events committed to be in the schedule
    loop $i = 1 \ldots n$   % Consider the events in sorted order.
        if( event $i$ does not conflict with an event in $Commit$ ) then
            $Commit = Commit \cup \{i\}$
    end loop
    return($Commit$)
end algorithm
```

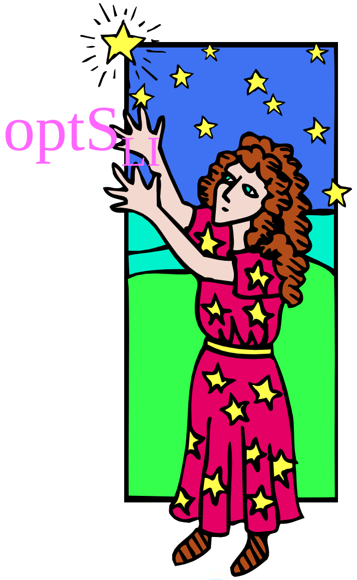
Massaging optSLI into optSours



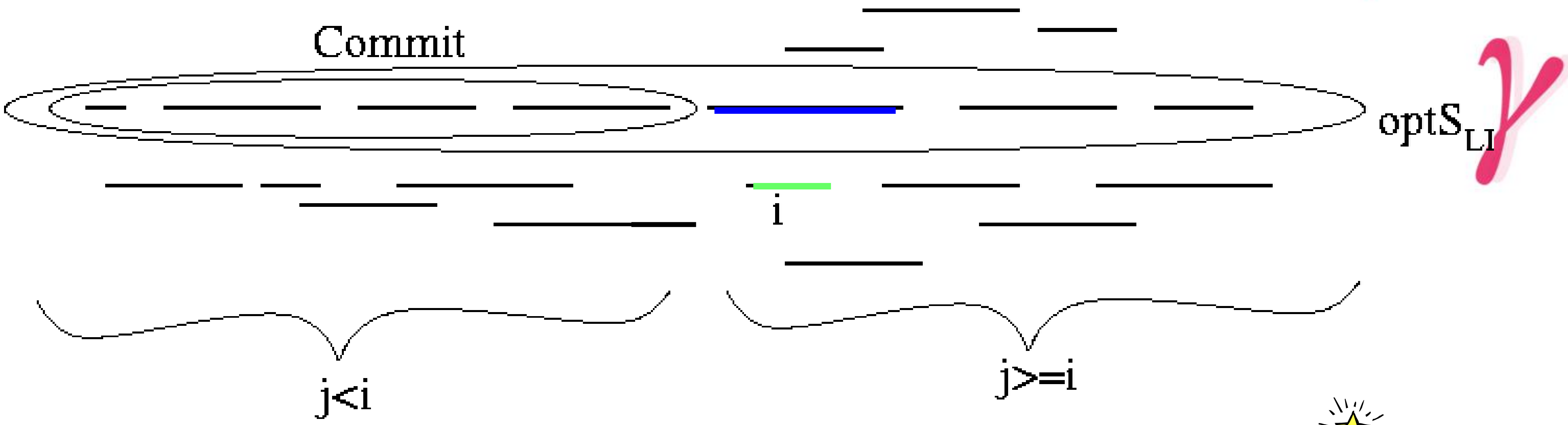


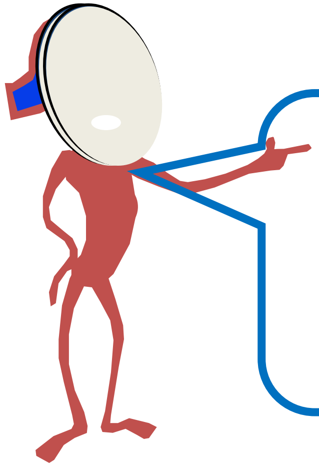
Start by adding new event i.

Delete events conflicting with job i.



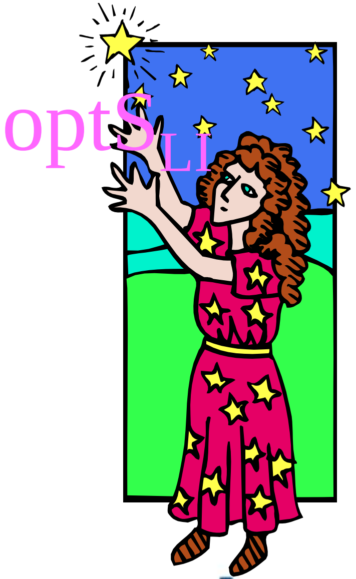
Massaging optSLI into optSours



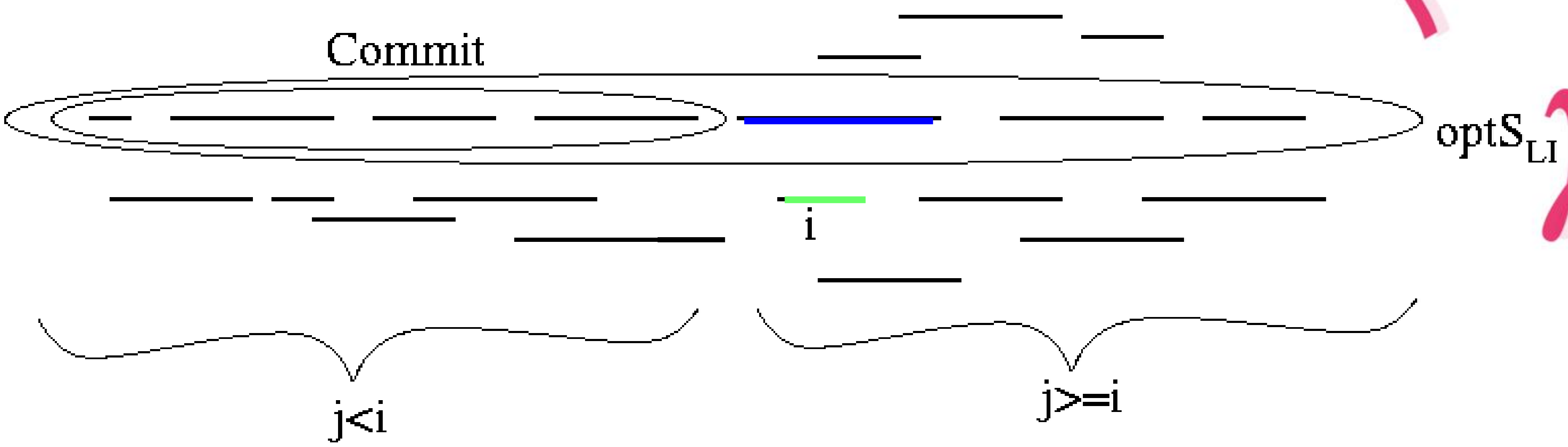


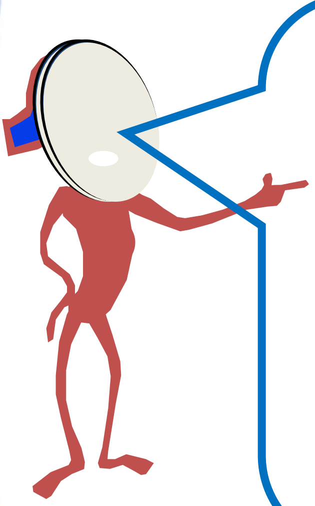
$\text{optS}_{\text{ours}}$ is valid

optS_{LI} was valid and we removed any new conflicts.



Massaging optSLI into optSours



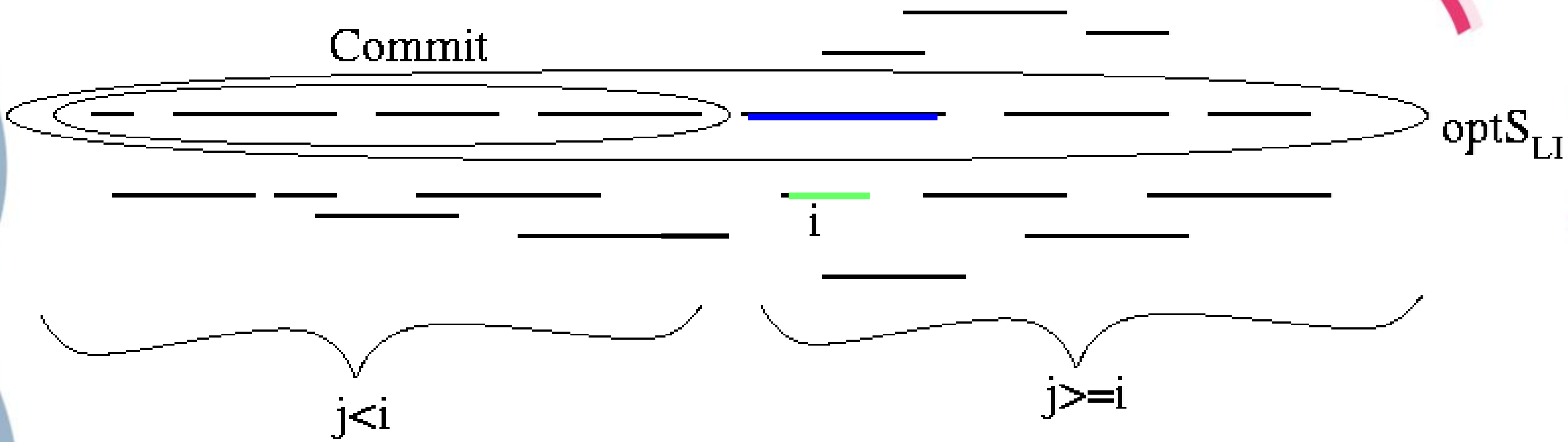


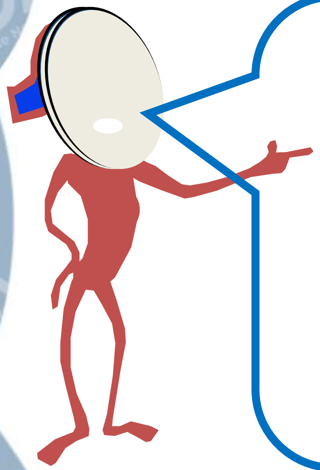
$\text{optS}_{\text{ours}}$ is consistent with our choices.

optS_{LI} was consistent with our prior choices.
We added event i.

Events in Commit don't conflict with event i and hence were not deleted.

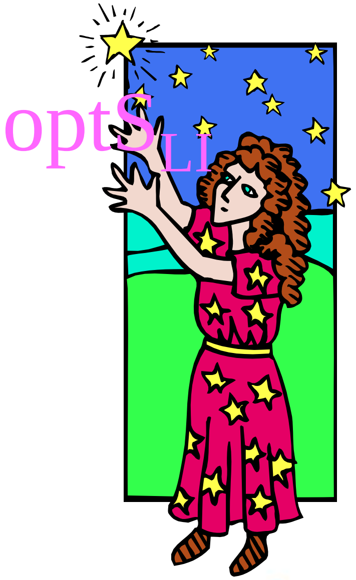
Massaging optSLI into optSours



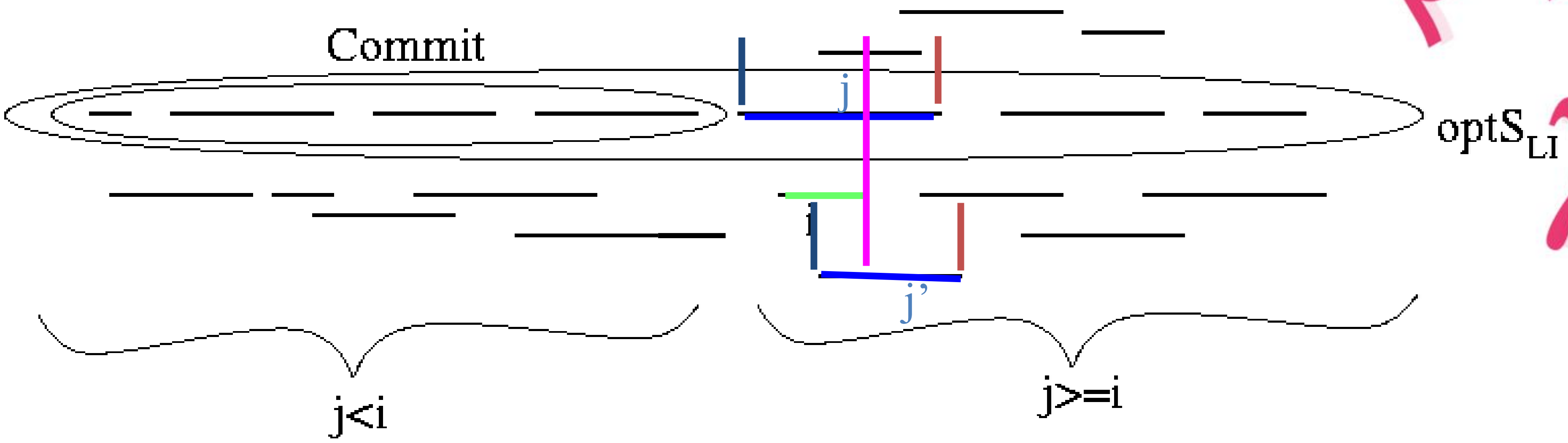


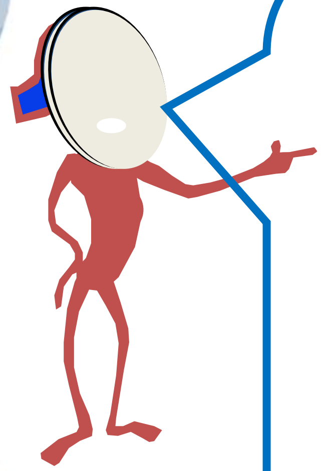
$\text{optS}_{\text{ours}}$ is optimal

optS_{LI} was optimal.
If we delete at most one event
then $\text{optS}_{\text{ours}}$ is optimal too.



Massaging optSLI into optSours





Deleted at most one event j

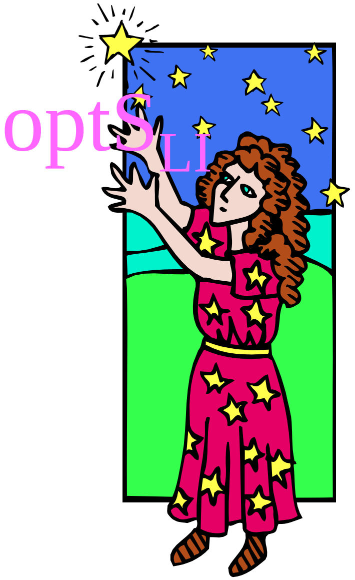
$i<j \;\; \Rightarrow f_i \leq f_j$

[j conflicts with i] Þ s_j £ f_i

Þ j runs at time f_i.

Two such j conflict with each other.

Only one in optS$_{LI}$.



Massaging optSLI into optSours

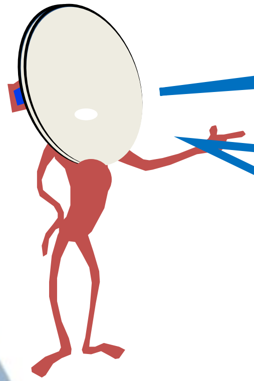
Case 1

$optS_{ours}$ is valid

$optS_{ours}$ is consistent

$optS_{ours}$ is optimal

$optS_{ours}$ → <LI>

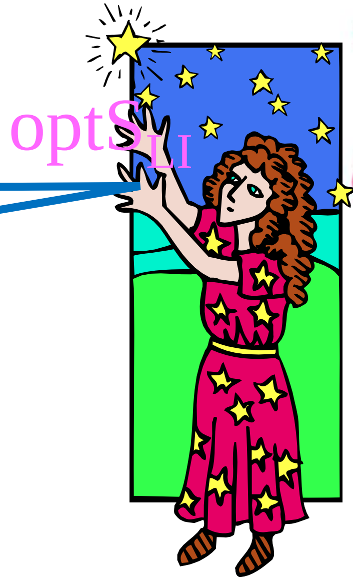


Maintaining Loop Invariant

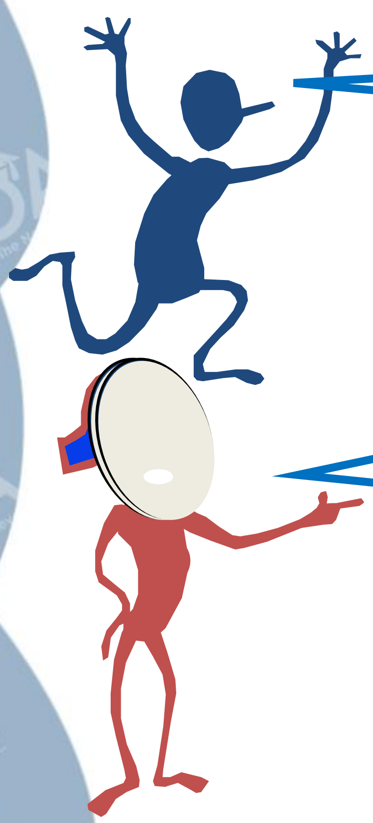
<LI>
¬<exit Cond>
codeB
→ <LI>

Case 2

Massaging optSLI into optSours



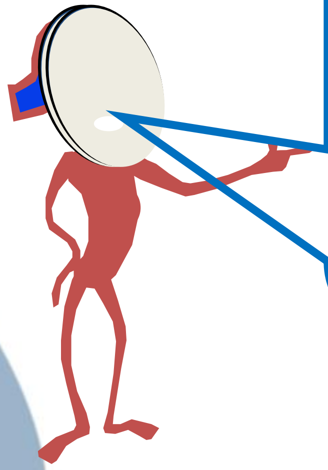
I hold optS_{LI} witnessing that there is an opt sol consistent with previous choices.



I reject next event i.

Event i conflicts with events committed to so it can’t be in optSLI either.

Massaging optSLI into optSours



Maintaining Loop Invariant

<LI>
¬<exit Cond>
codeB ⇒ <LI>

Clean up loose ends

<loop-invariant>
<exit Cond>
codeC
→ <postCond>

<exit Cond> → Alg commits to or reject each event.

Has a solution S.

<LI> → ∃ opt sol consistent with these choices.

S must be optimal.

codeC → Alg returns optS .

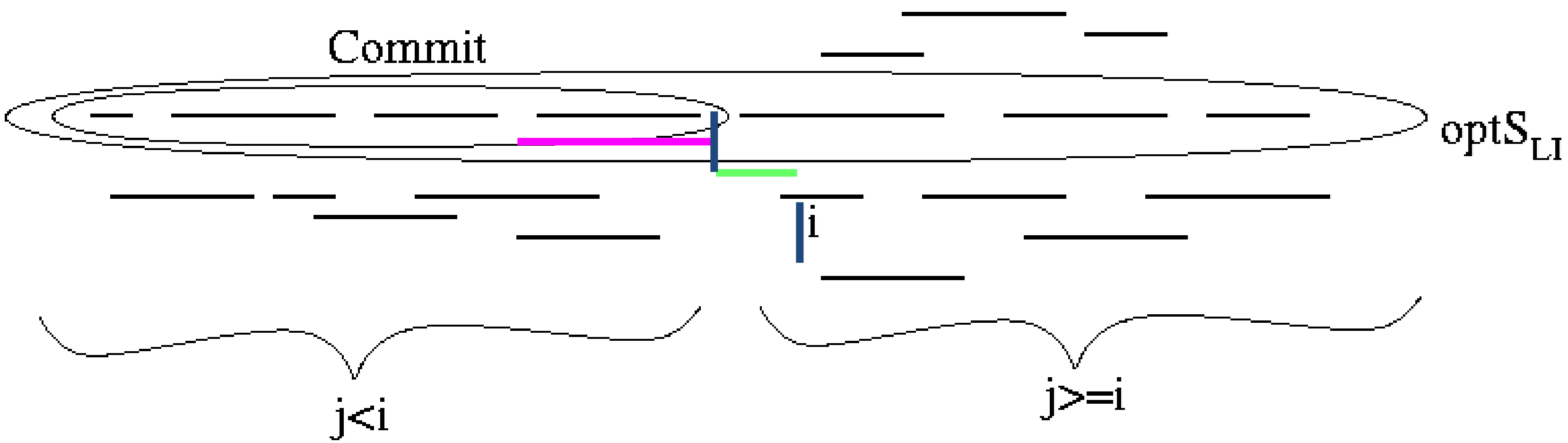
→ <postCond>

Running Time

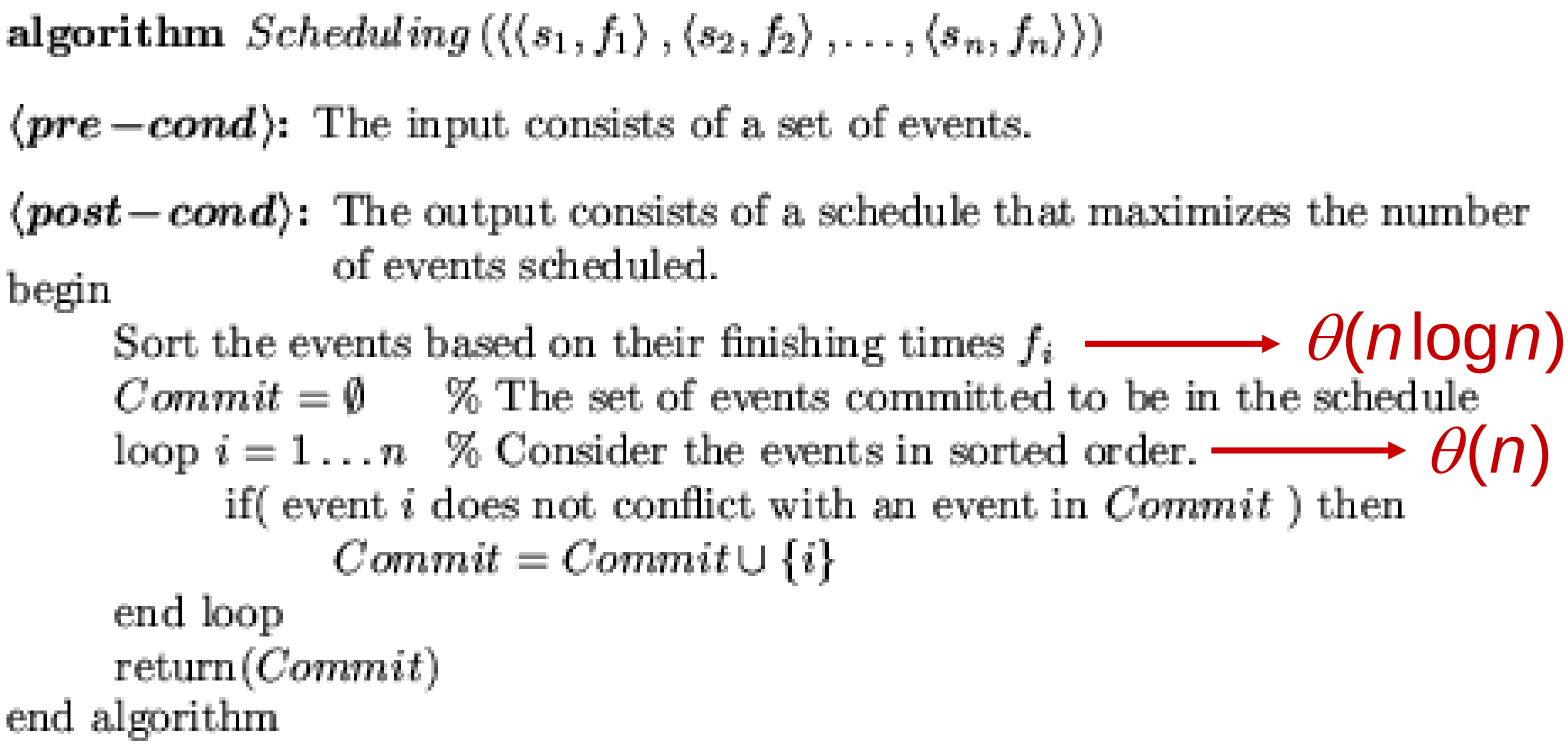
Greedy algorithms are very fast because they only consider each object once.

Checking whether next event i conflicts with previously committed events requires

only comparing it with the last such event.



Running Time



```
algorithm $Scheduling(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle)$
$\langle pre-cond \rangle$: The input consists of a set of events.
$\langle post-cond \rangle$: The output consists of a schedule that maximizes the number
                  of events scheduled.
begin
    Sort the events based on their finishing times $f_i$
    $Commit = \emptyset$     % The set of events committed to be in the schedule
    loop $i = 1 \ldots n$   % Consider the events in sorted order.
        if( event $i$ does not conflict with an event in $Commit$ ) then
            $Commit = Commit \cup \{i\}$
    end loop
    return($Commit$)
end algorithm
```

$\longrightarrow \theta(n\log n)$

$\longrightarrow \theta(n)$

$\to T(n) = \theta(n\log n)$

Example 3: Minimum Spanning Trees

Minimum Spanning Trees

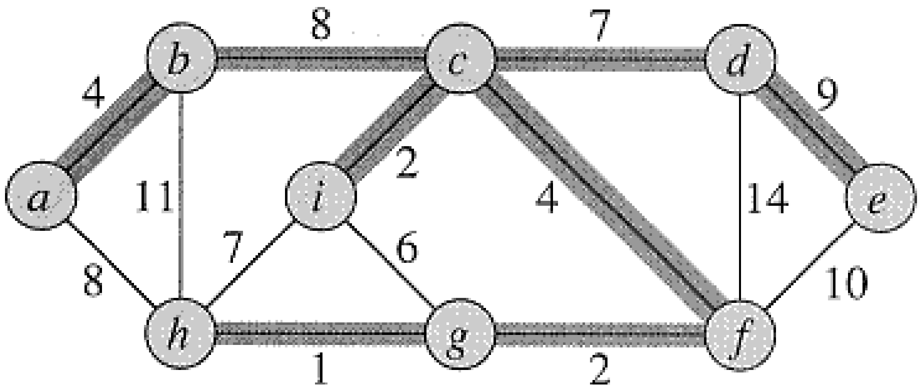
Example Problem

You are planning a new terrestrial telecommunications network to connect a number of remote mountain villages in a developing country.

The cost of building a link between pairs of neighbouring villages *(u,v)* has been estimated: *w(u,v).*

You seek the minimum cost design that ensures each village is connected to the network.

The solution is called a *minimum spanning tree (MST)*.



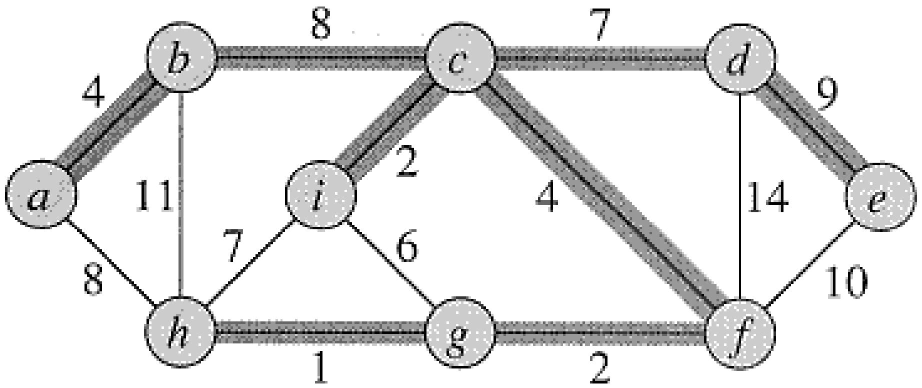
Minimum Spanning Trees

The problem is defined for any undirected, connected, weighted graph.

The weight of a subset T of a weighted graph is defined as:

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Thus the MST is the spanning tree T that minimizes $w(T)$



Building the Minimum Spanning Tree

Iteratively construct the set of edges *A* in the MST.

Initialize *A* to {}

As we add edges to *A*, maintain a Loop Invariant:

A is a subset of some MST

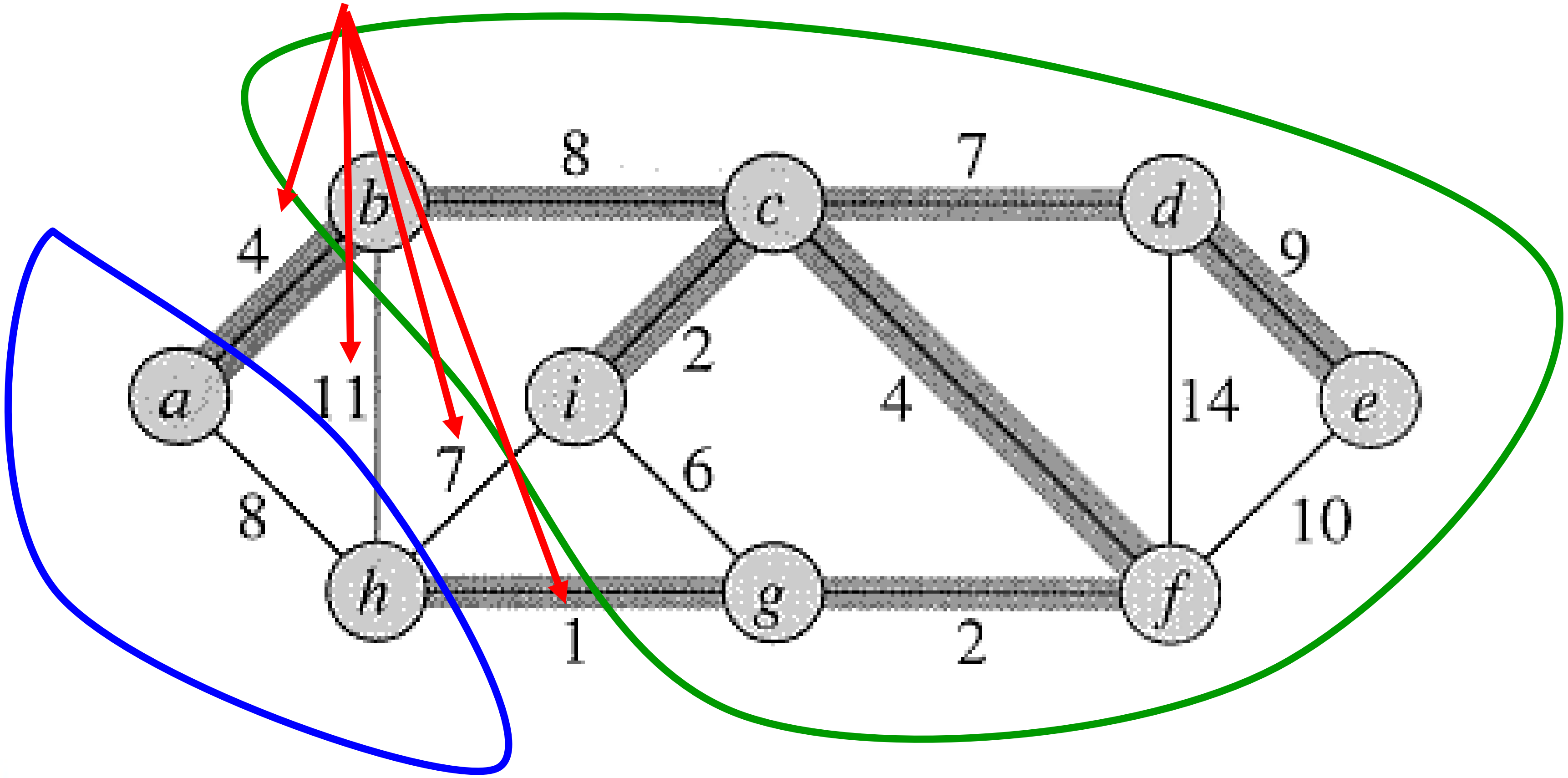
Maintain loop invariant and make progress by only adding **safe** edges.

An edge *(u,v)* is called ***safe*** for *A* iff AÈ({u,v}) is also a subset of some MST.

Finding a safe edge

Idea: Every 2 disjoint subsets of vertices must be connected by at least one edge.

Which one should we choose?



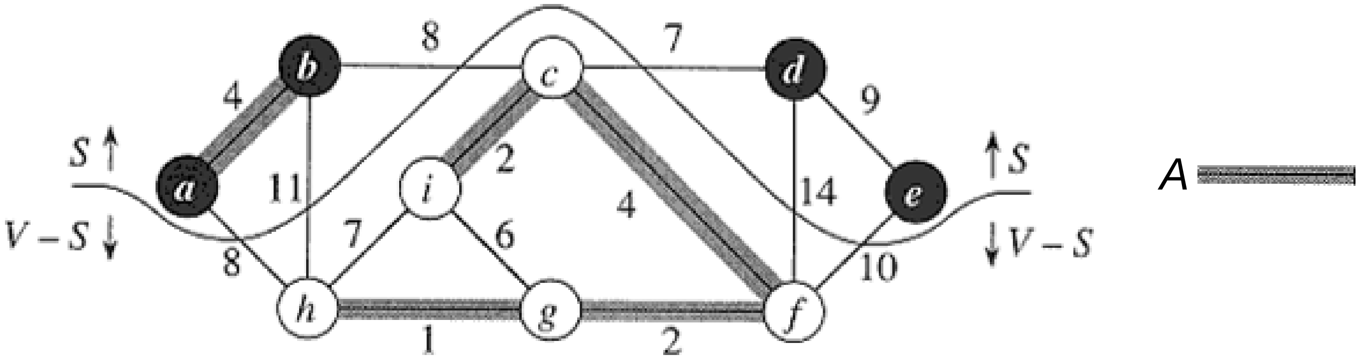
Some definitions

A *cut* *(S,V-S)* is a partition of vertices into disjoint sets *S* and *V-S*.

Edge *(u,v)*$\in$*E* *crosses* cut *(S, V-S)* if one endpoint is in *S* and the other is in *V-S*.

A cut *respects* a set of edges *A* iff no edge in *A* crosses the cut.

An edge is a *light* edge crossing a cut iff its weight is minimum over all edges crossing the cut.



Minimum Spanning Tree Theorem

Let

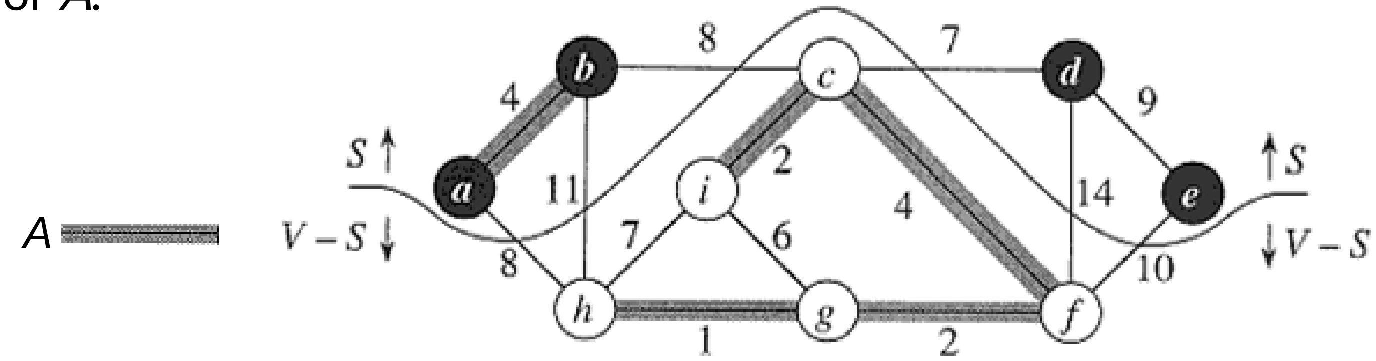
A be a subset of some MST

(S,V-S) be a cut that respects *A*

(u,v) be a **light** edge crossing *(S,V-S)*

Then

(u,v) is safe for *A*.



Basis for a greedy algorithm

Proof

Let *G* be a connected, undirected, weighted graph.

Let *T* be an MST that includes *A*.

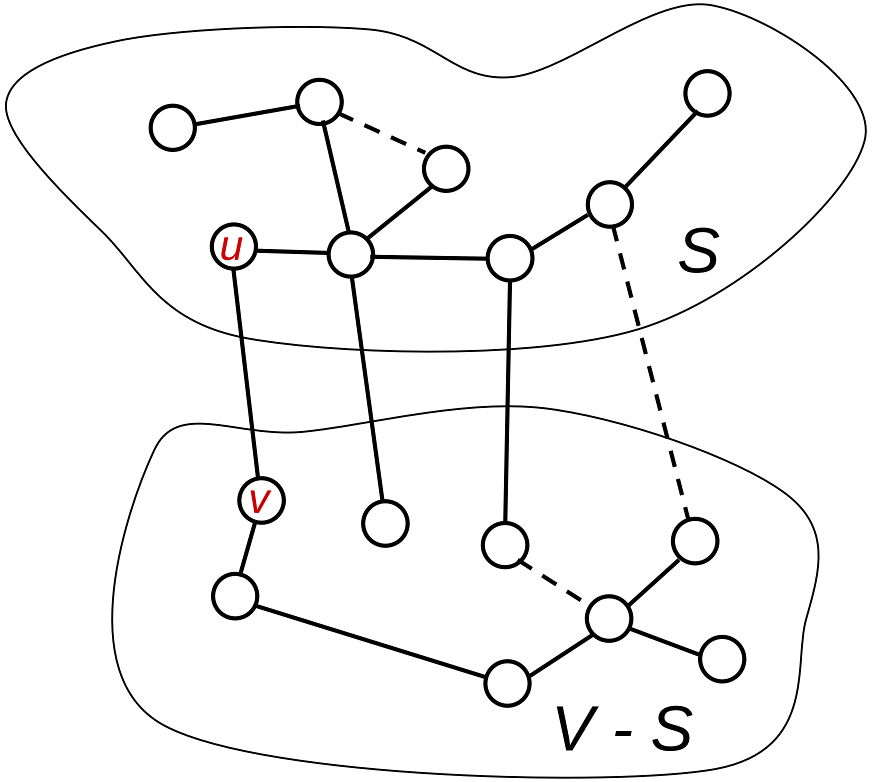
Let (*S*, *V*-*S*) be a cut that respects *A*.

Let *(u,v)* be a light edge between *S* and *V*-*S*.

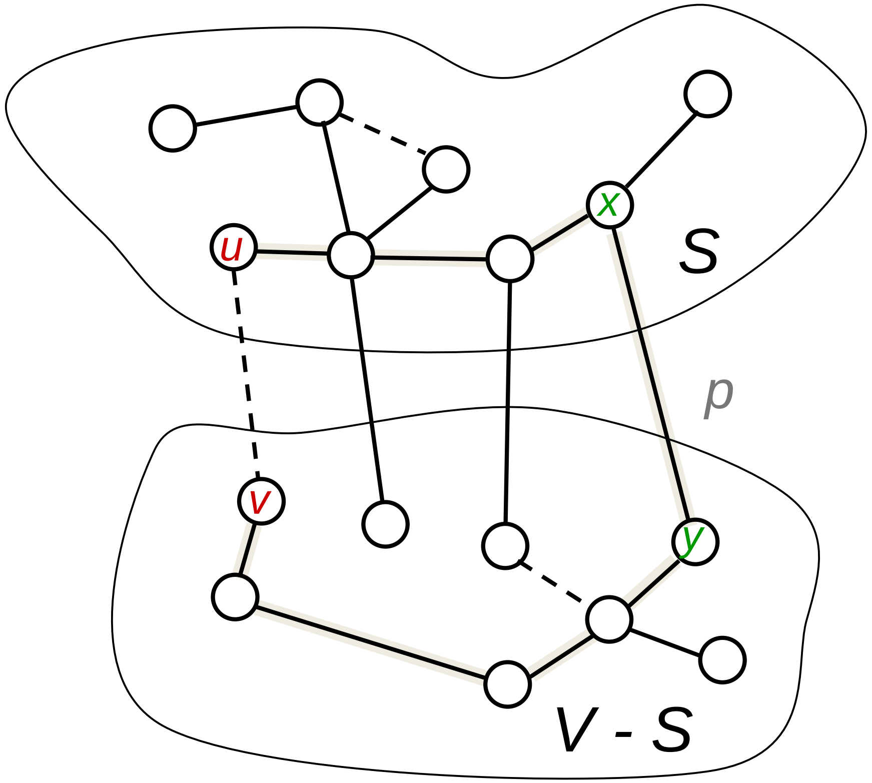
If *T* contains *(u,v)* then we're done.

——— Edge $\in T$

- - - - - Edge $\notin T$



- Suppose *T* does not contain *(u,v)*
  - ✓ Can construct different MST *T′* that includes *AÈ(u,v)*
  - ✓ The edge *(u,v)* forms a cycle with the edges on the path *p* from *u* to *v* in *T*.
  - ✓ There is at least one edge in *p* that crosses the cut: let that edge be *(x,y)*
  - ✓ *(x,y)* is not in *A*, since the cut *(S,V-S)* respects *A*.
  - ✓ Form new spanning tree *T′* by deleting *(x,y)* from *T* and adding *(u,v)*.
  - ✓ *w(T′)* ≤ *w(T)*, since *w(u,v)* ≤ *w(x,y)* → *T′* is an MST.
  - ✓ *A* ⊆ *T′*, since *A* ⊆ *T* and *(x,y)*∉*A* → *AÈ(u,v)* ⊆ *T′*
  - ✓ Thus *(u,v)* is safe for *A*.



End of Lecture 17

Kruskal’s Algorithm for computing MST

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that crosses the cut between them.
- Scans the set of edges in monotonically increasing order by weight (greedy).

Kruskal’s Algorithm: Loop Invariant

Let $A =$ solution under construction.

Let $E_i =$ the subset of i lowest-weight edges thus far considered

$<$ loop-invariant $>$:

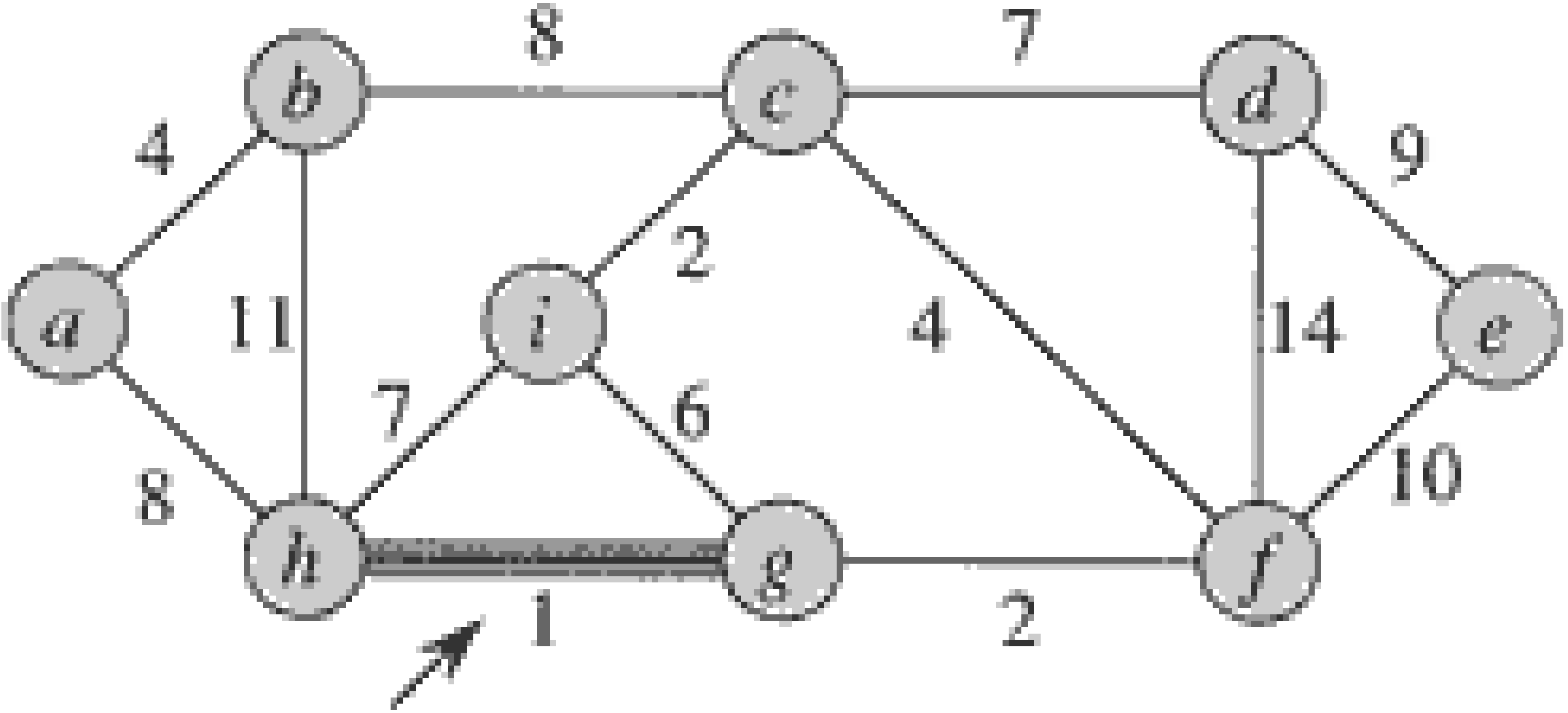
$\exists$ MST T :

1) $A \in T$,

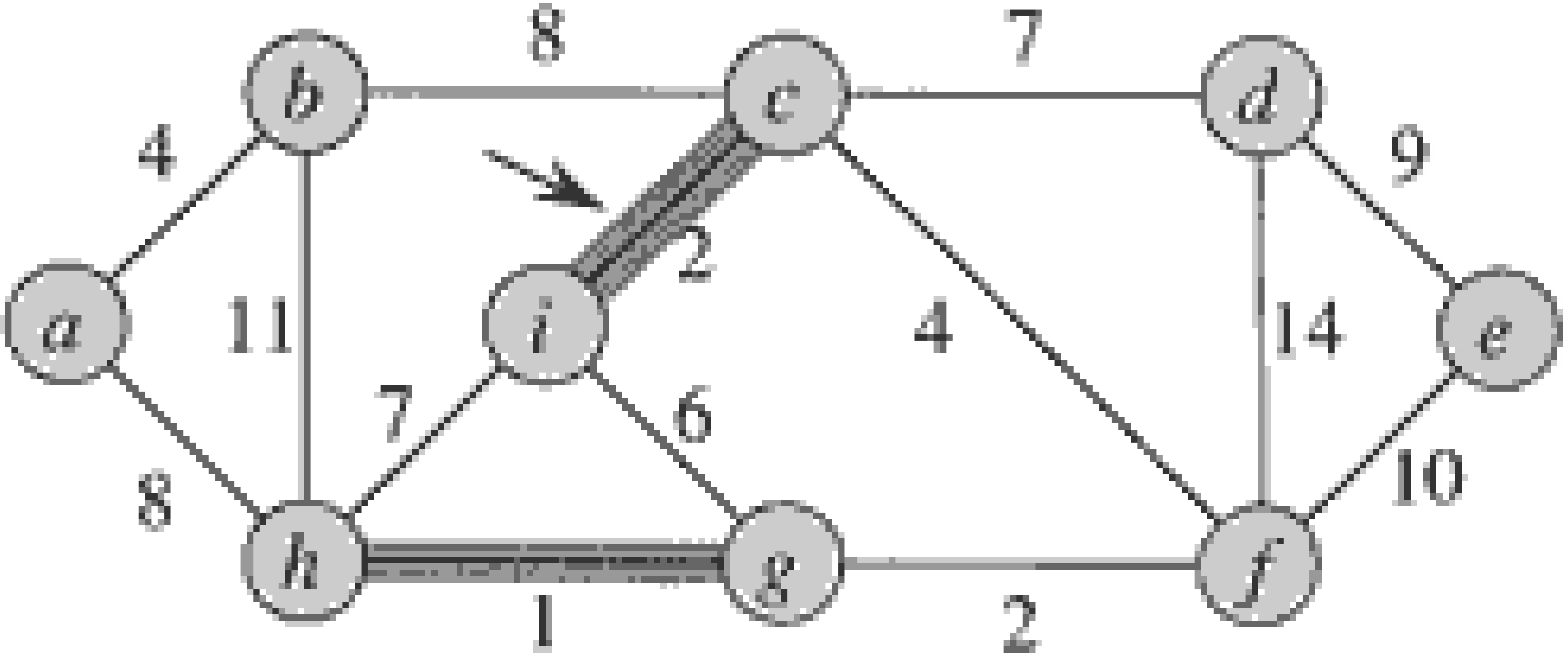
2) $\forall (u,v) \in E_i$:

$(u,v) \in A$ or $(u,v) \notin T$

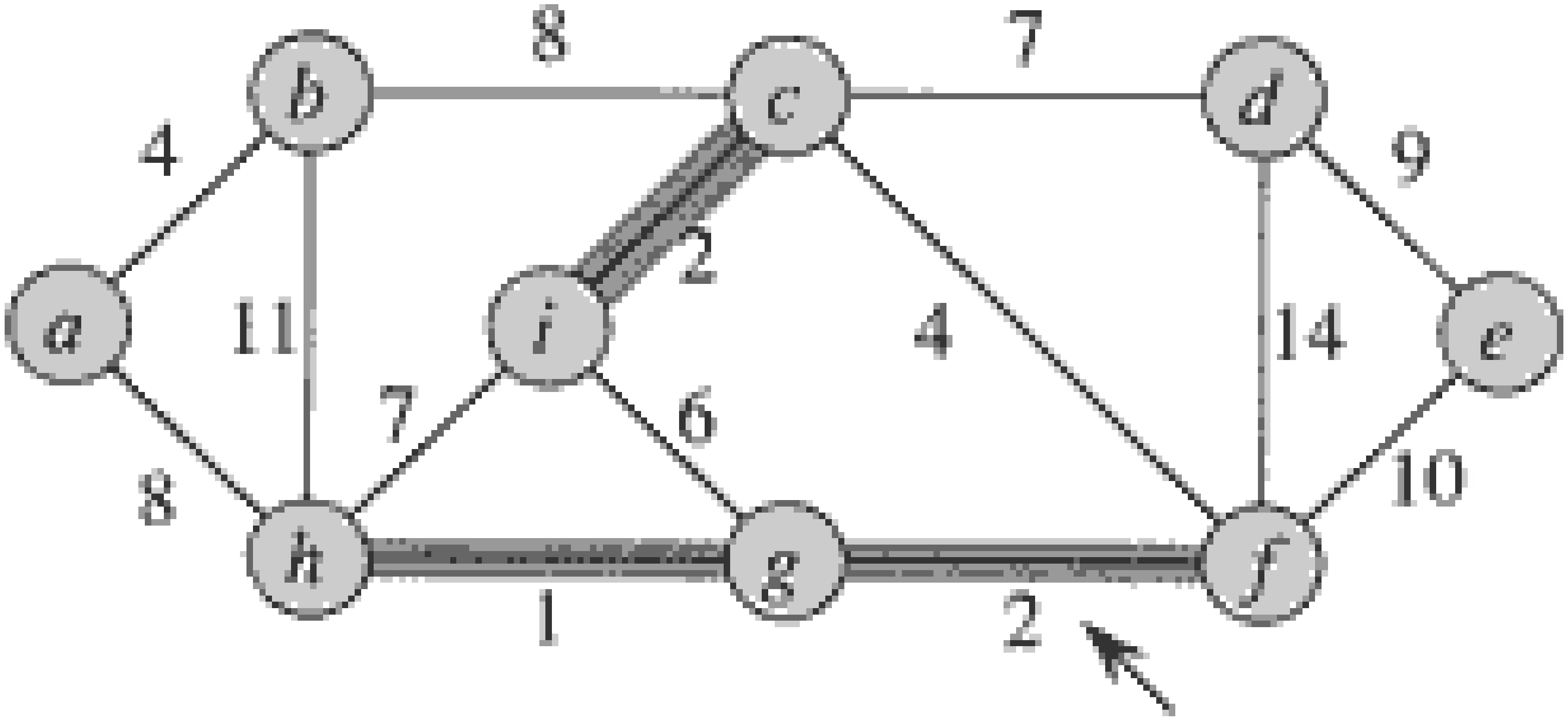
Kruskal’s Algorithm: Example



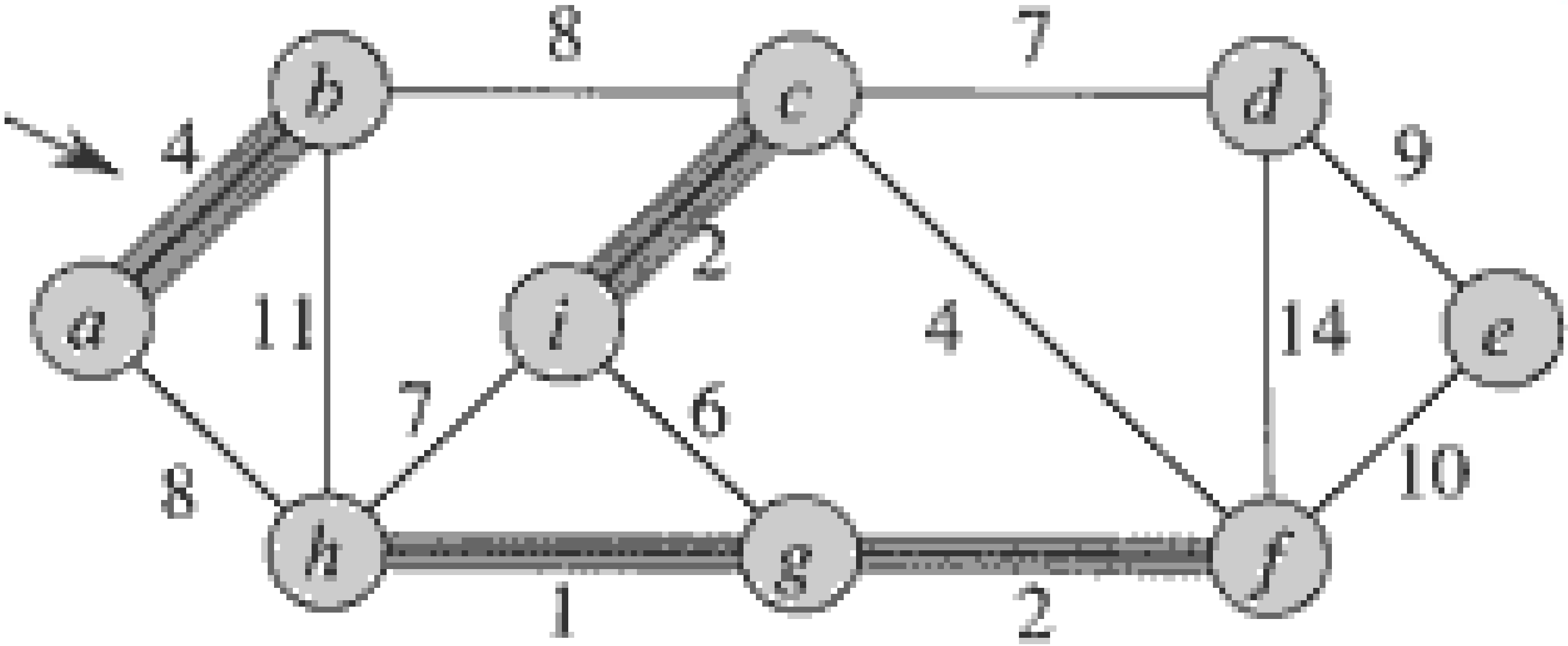
Kruskal’s Algorithm: Example



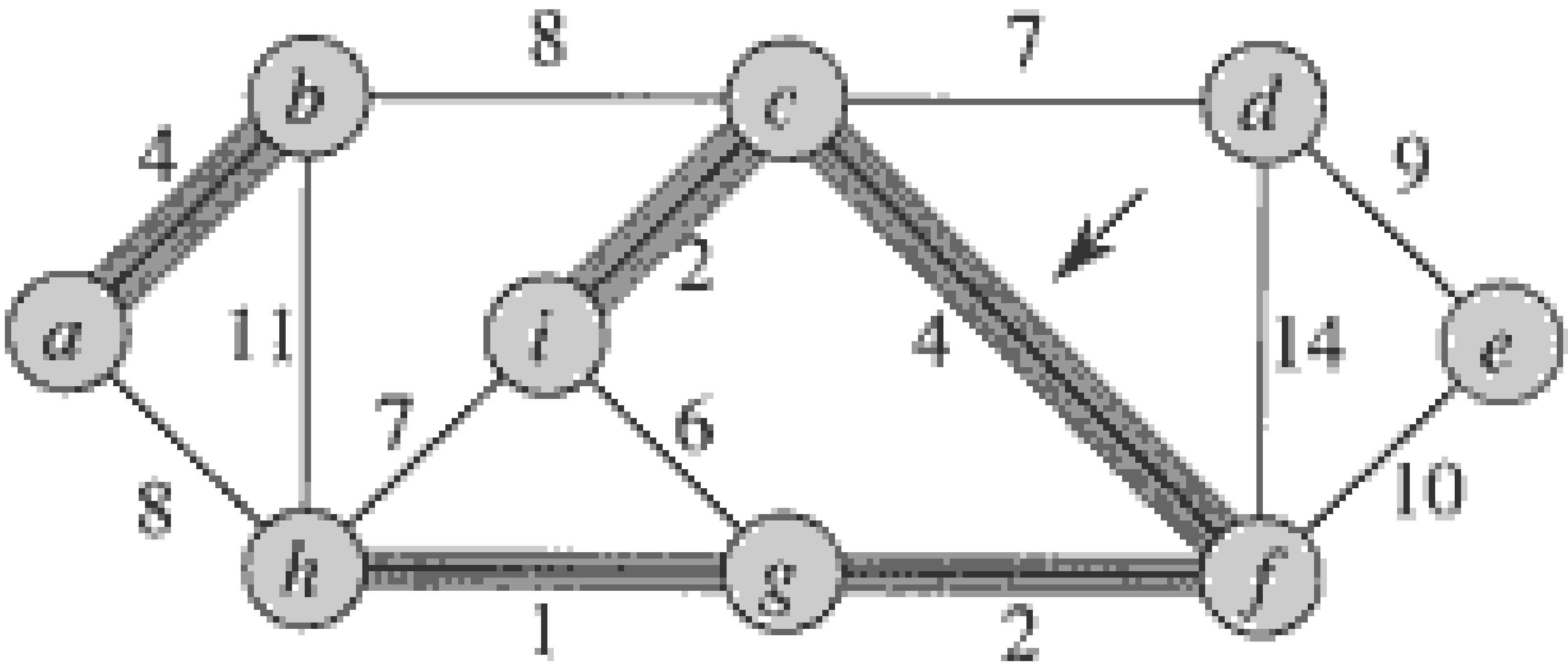
Kruskal’s Algorithm: Example



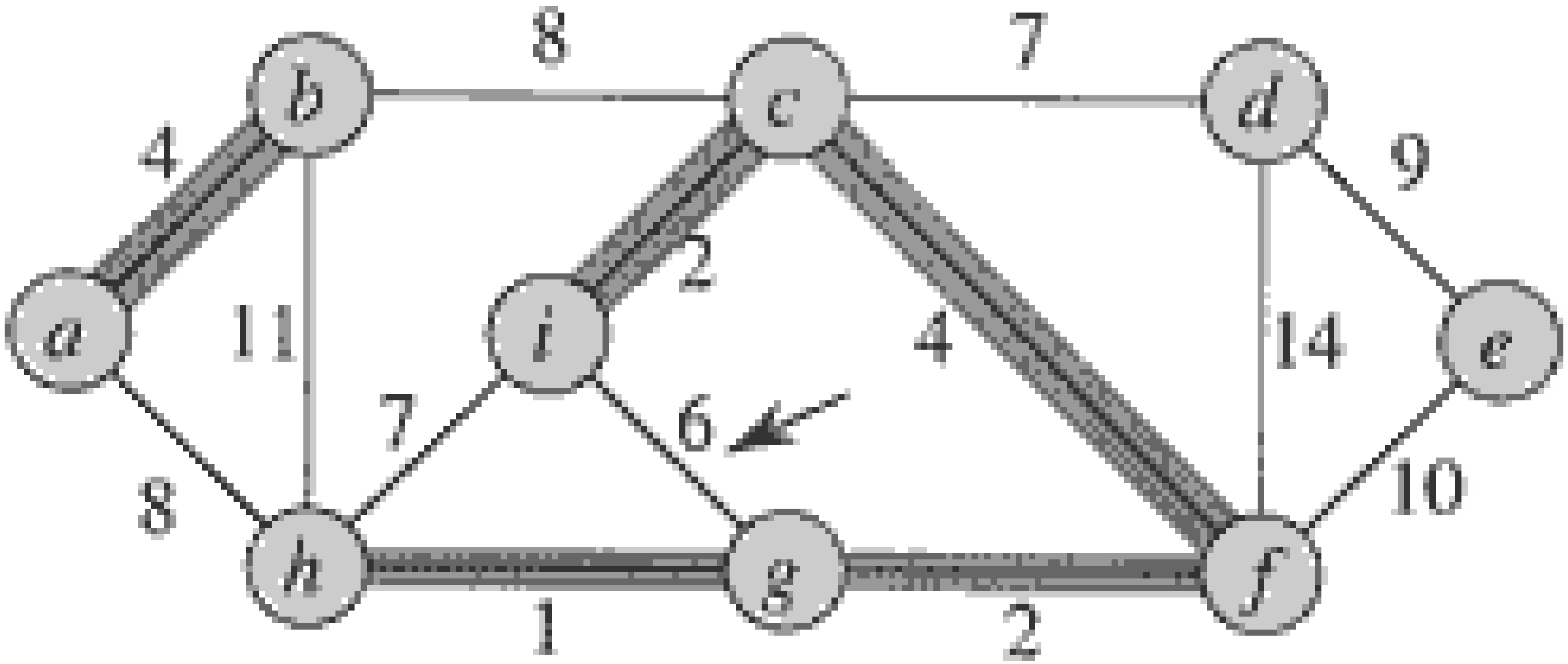
Kruskal's Algorithm: Example



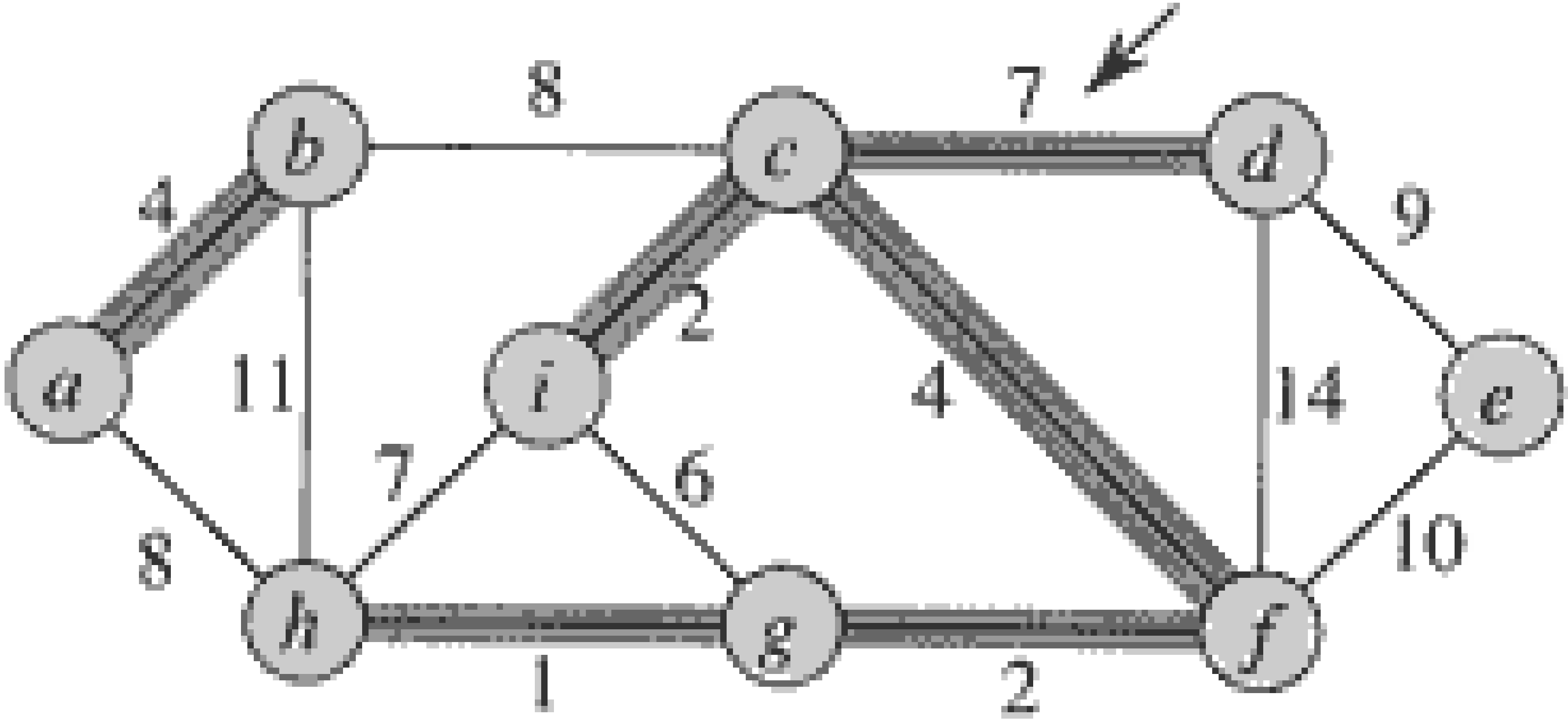
Kruskal’s Algorithm: Example



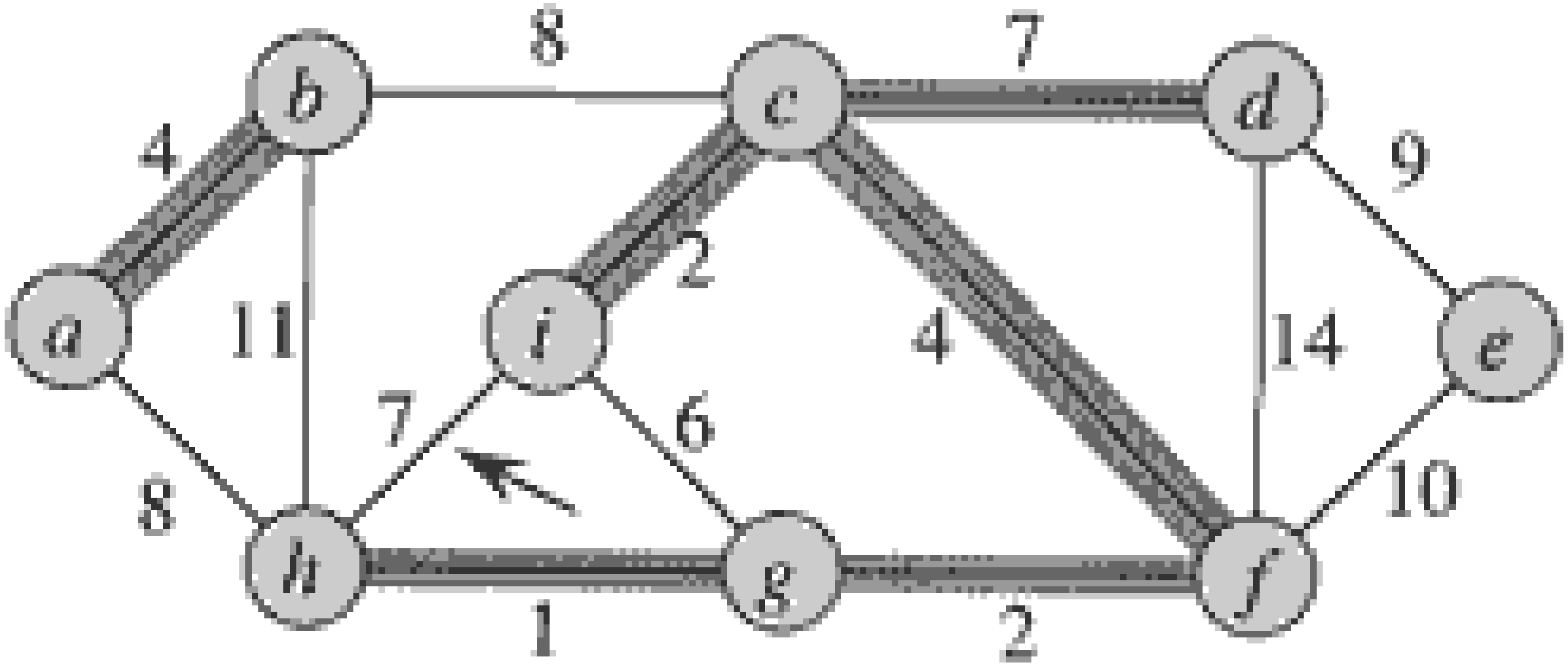
Kruskal's Algorithm: Example



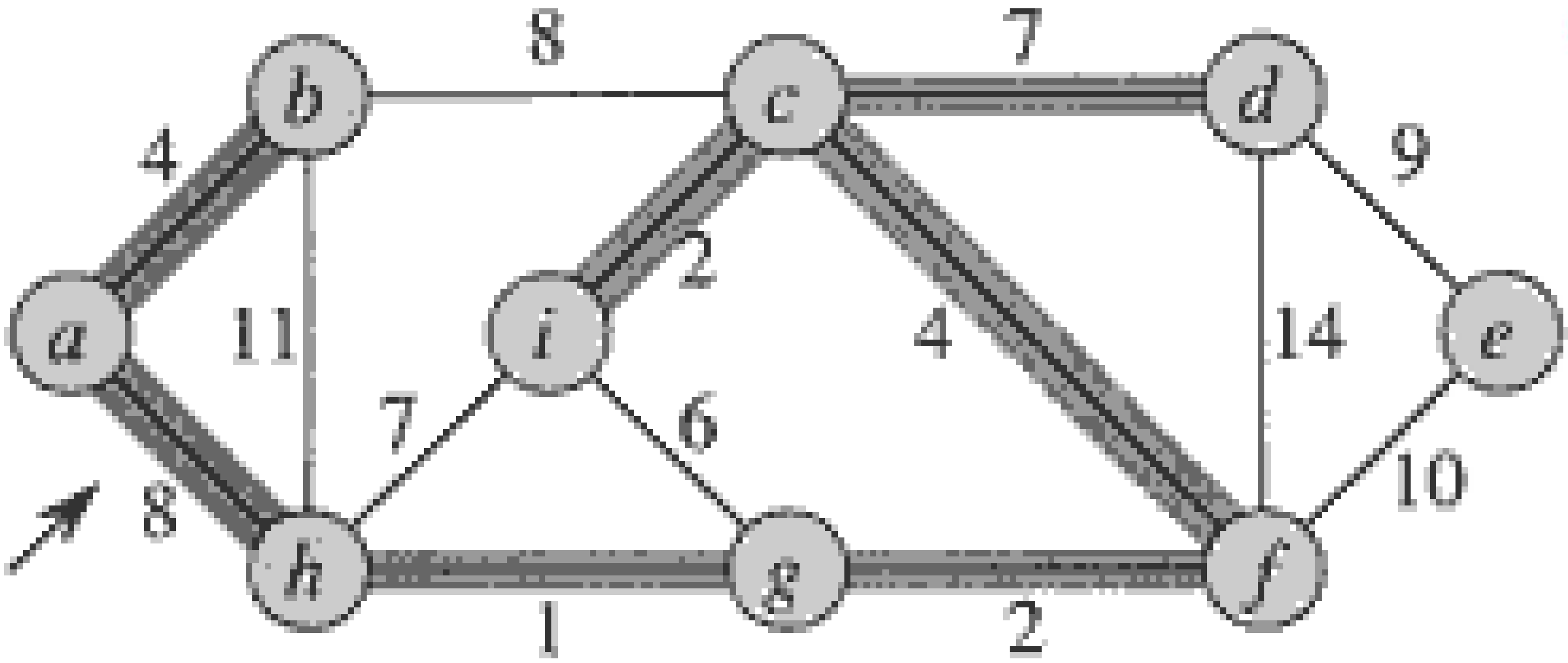
Kruskal’s Algorithm: Example



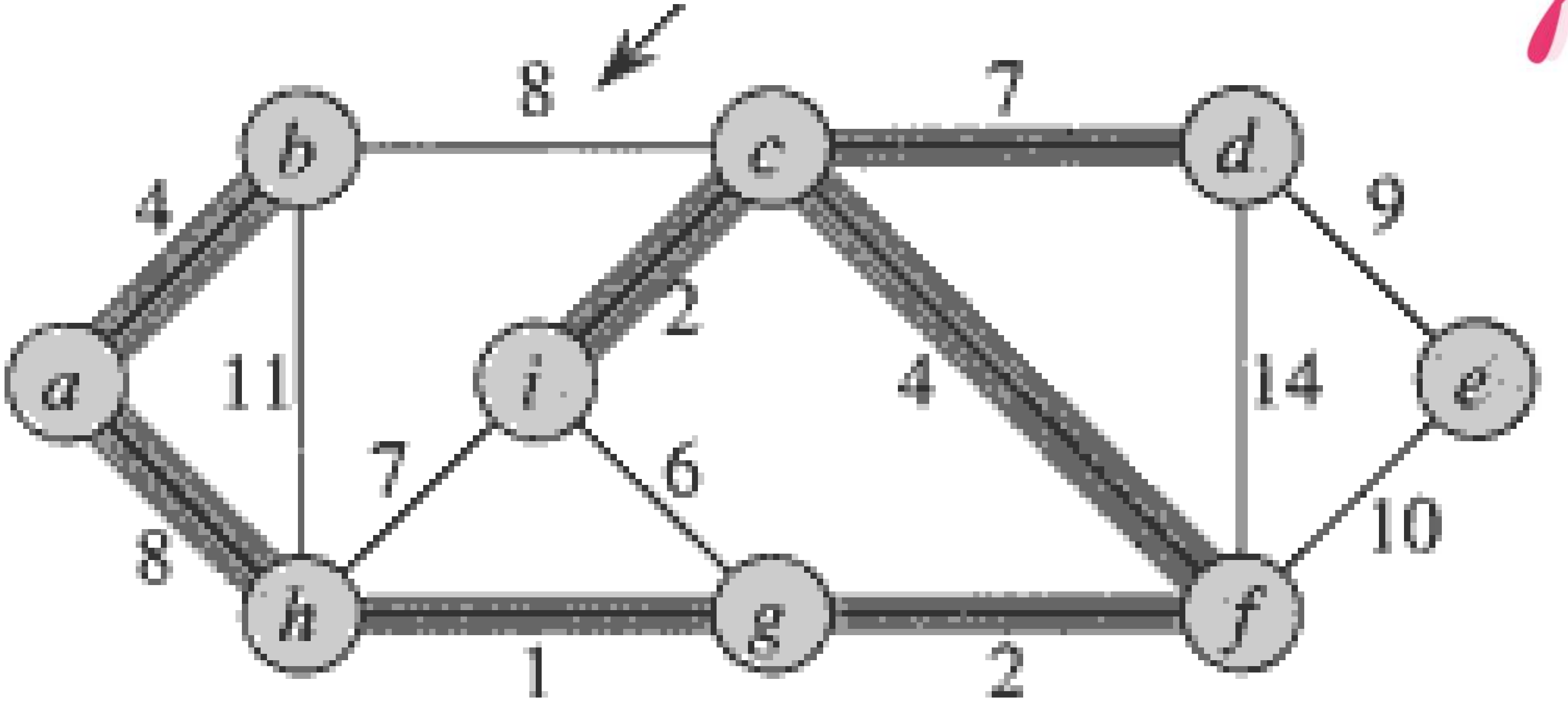
Kruskal’s Algorithm: Example



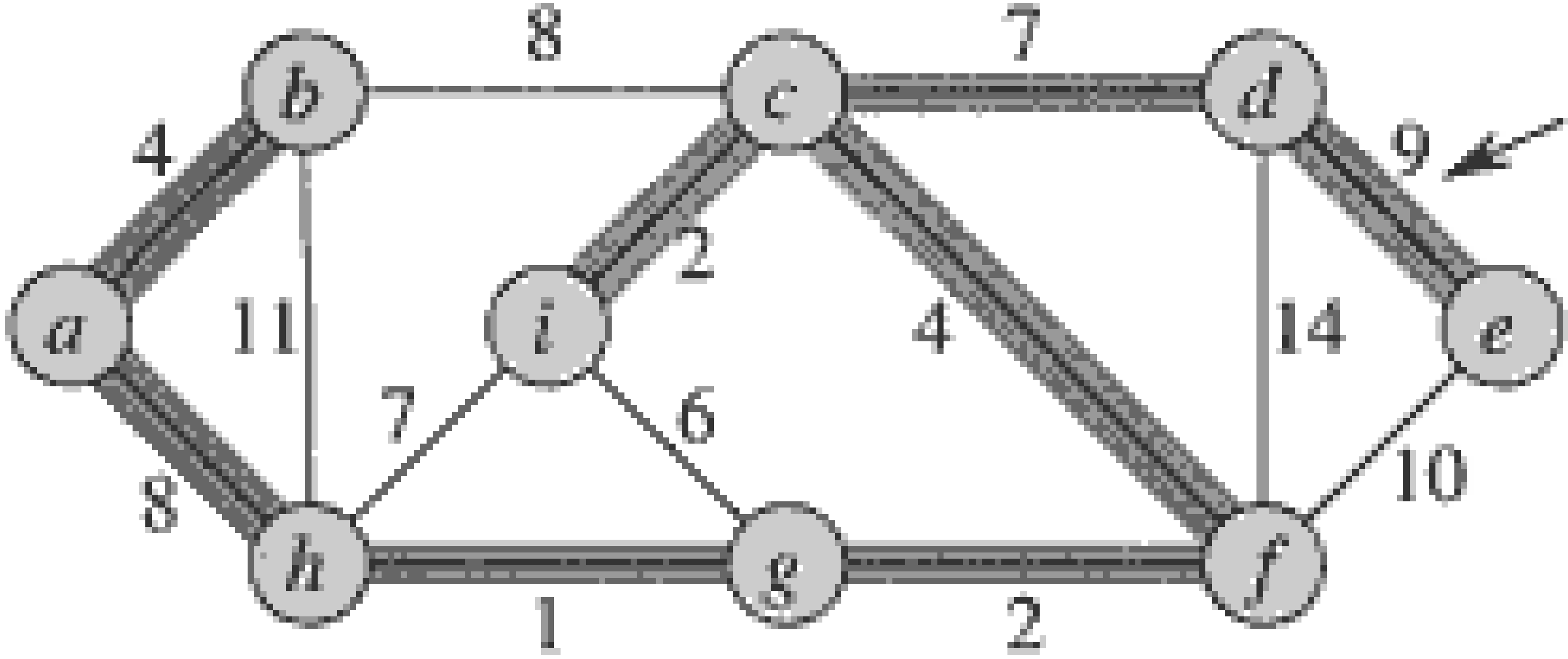
Kruskal’s Algorithm: Example



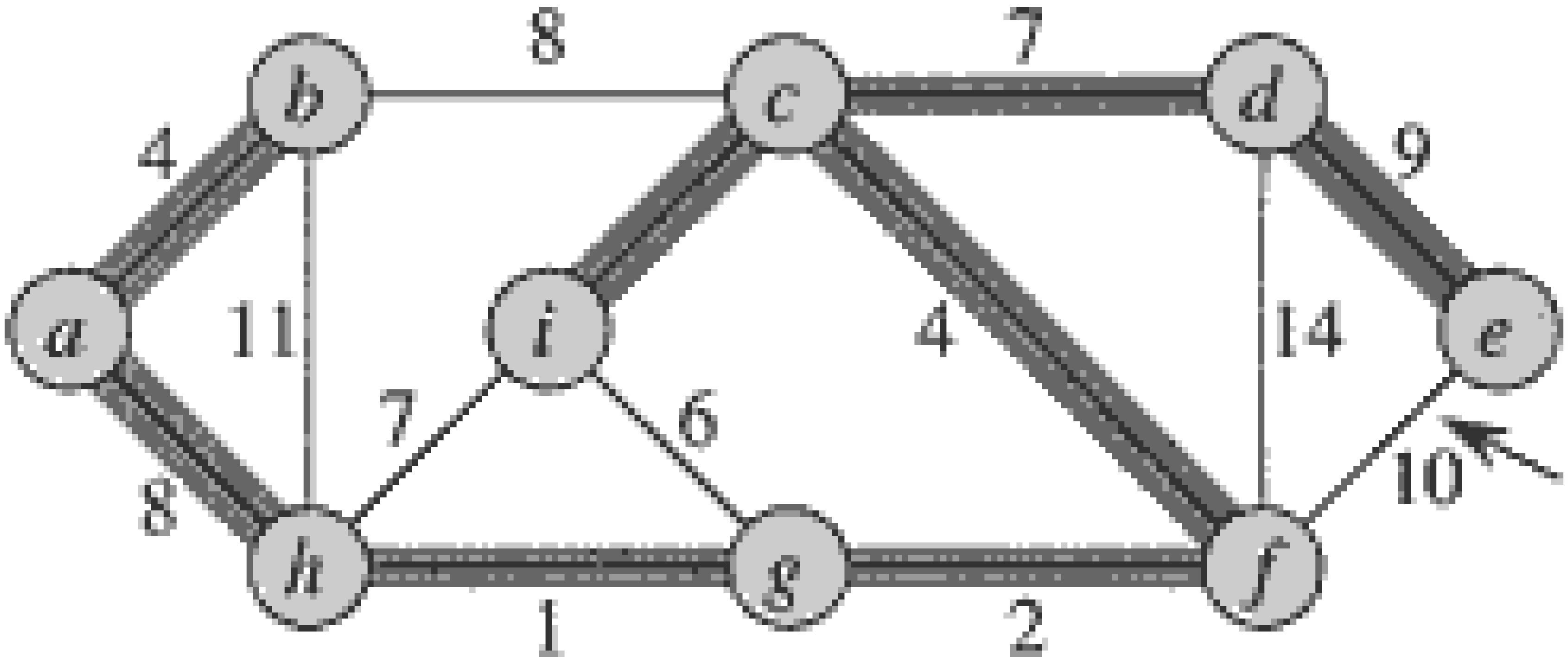
Kruskal’s Algorithm: Example



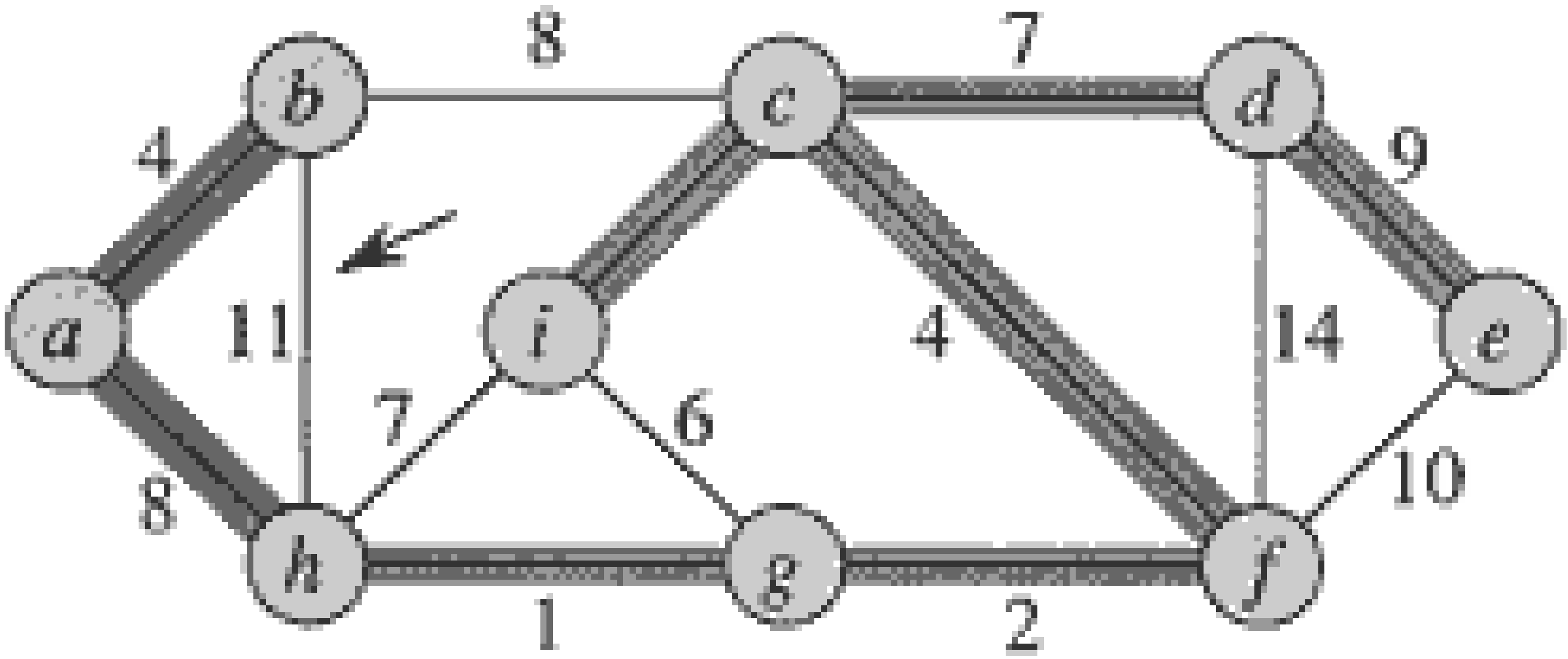
Kruskal’s Algorithm: Example



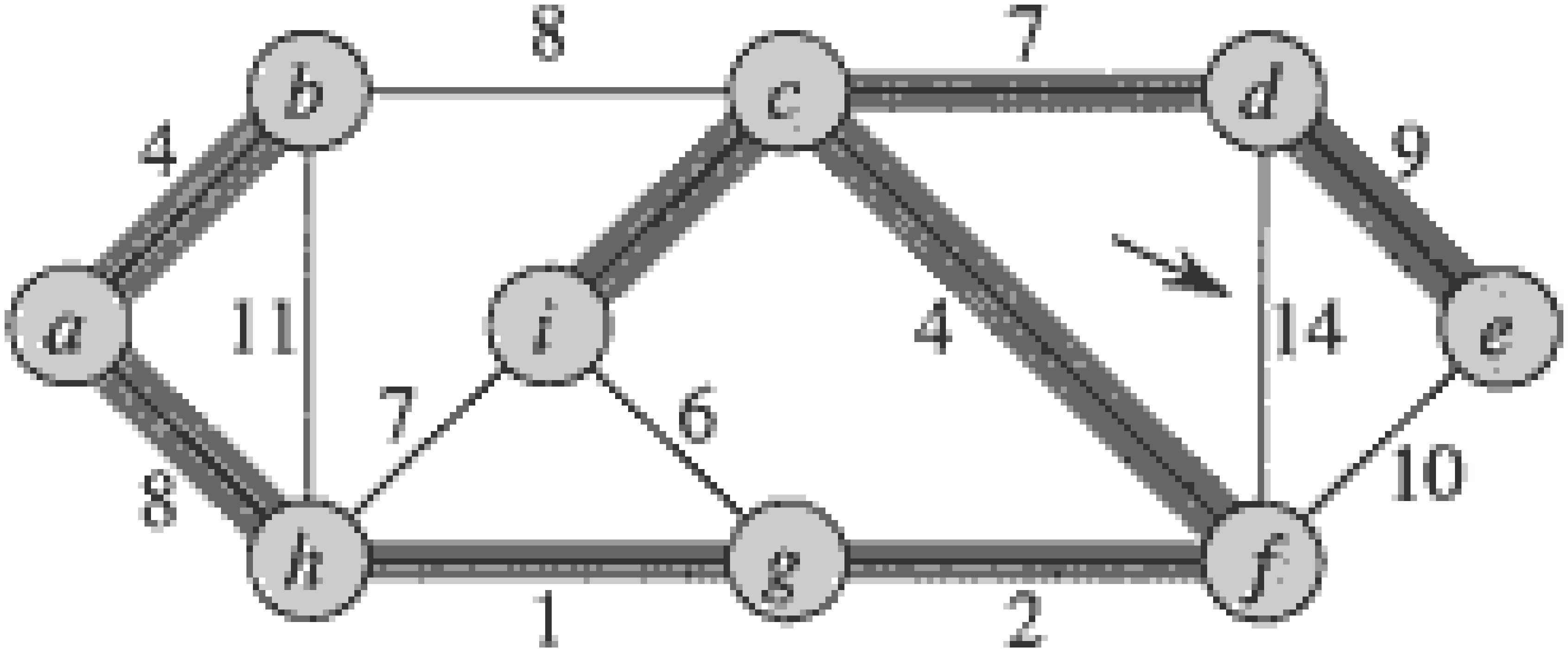
Kruskal’s Algorithm: Example



Kruskal’s Algorithm: Example



Kruskal’s Algorithm: Example



Finished!

Disjoint Set Data Structures

- Disjoint set data structures can be used to represent the disjoint connected components of a graph.

- Make-Set(*x*) makes a new disjoint component containing only vertex *x*.

- Union(*x*,*y*) merges the disjoint component containing vertex x with the disjoint component containing vertex *y*.

- Find-Set(*x*) returns a vertex that represents the disjoint component containing *x*.

Disjoint Set Data Structures

- Most efficient representation represents each disjoint set (component) as a tree.
- Time complexity of a sequence of m operations, n of which are Make-Set operations, is:

$$O(m \times \alpha(n))$$

where $\alpha(n)$ is Ackerman's function, which grows extremely slowly.

n	$\alpha(n)$
3	1
7	2
2047	3
10^{80}	4

Kruskal’s Algorithm for computing MST

```
Kruskal($G$,$w$)
$A = \varnothing$
for each vertex $v \in V[G]$
    Make-Set(v)
sort E[G] into nondecreasing order: $E[1...n]$
for $i = 1 : n$
$<$ loop-invariant $>$:
$\exists$ MST $T$ : 1) $A \in T$,
            2) $\forall (u,v) \in E[1...i-1]$: $(u,v) \in A$ or $(u,v) \notin T$
    $(u,v) = E[i]$
    if Find-Set($u$) $\neq$ Find-Set($v$)
        $A = A \cup \{(u,v)\}$
        Union($u$,$v$)
return $A$
```

Running Time = $O(E\log E)$

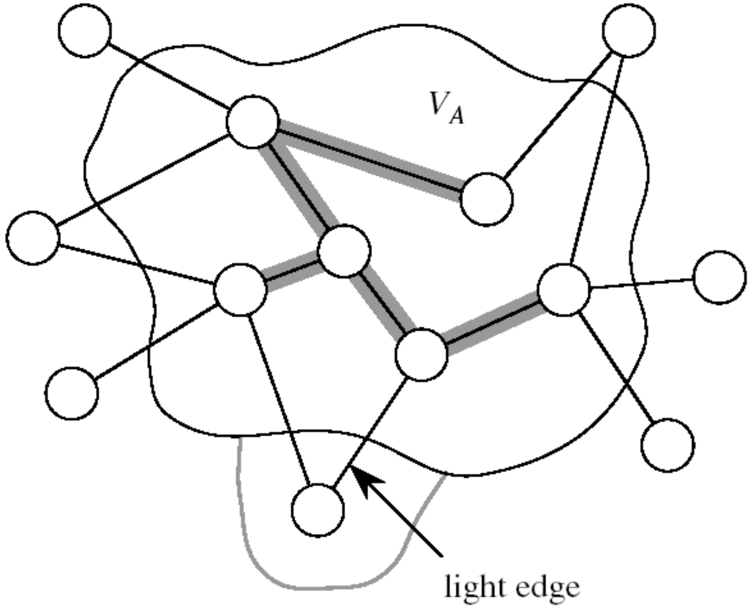
$= O(E\log V)$

Prim’s Algorithm for Computing MST

Build one tree A

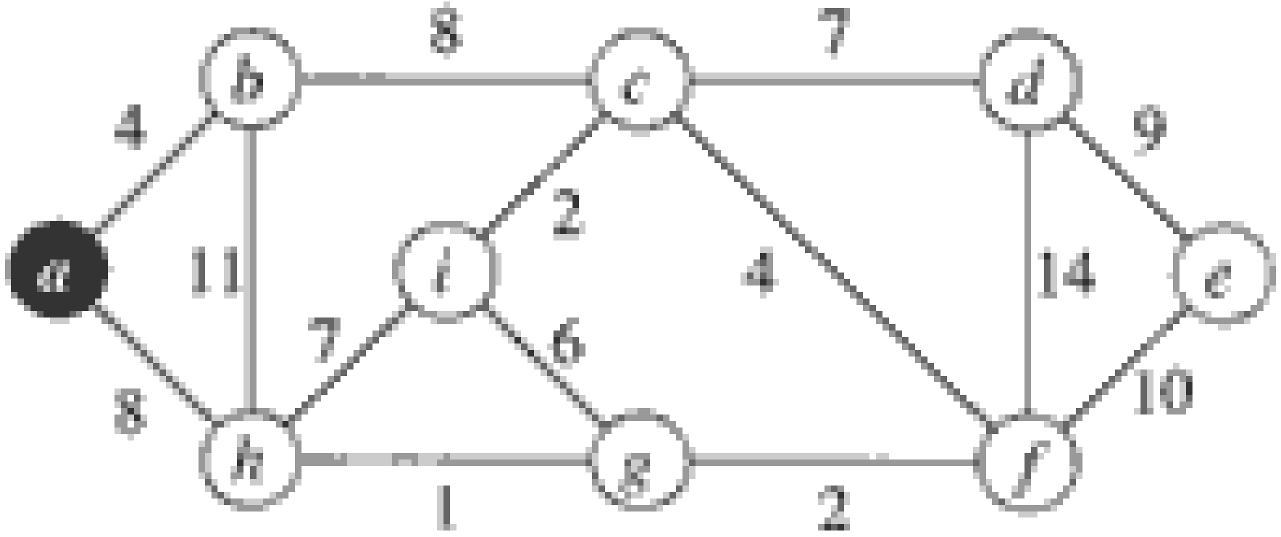
Start from arbitrary root r

At each step, add light edge connecting V_A to V- V_A (greedy)

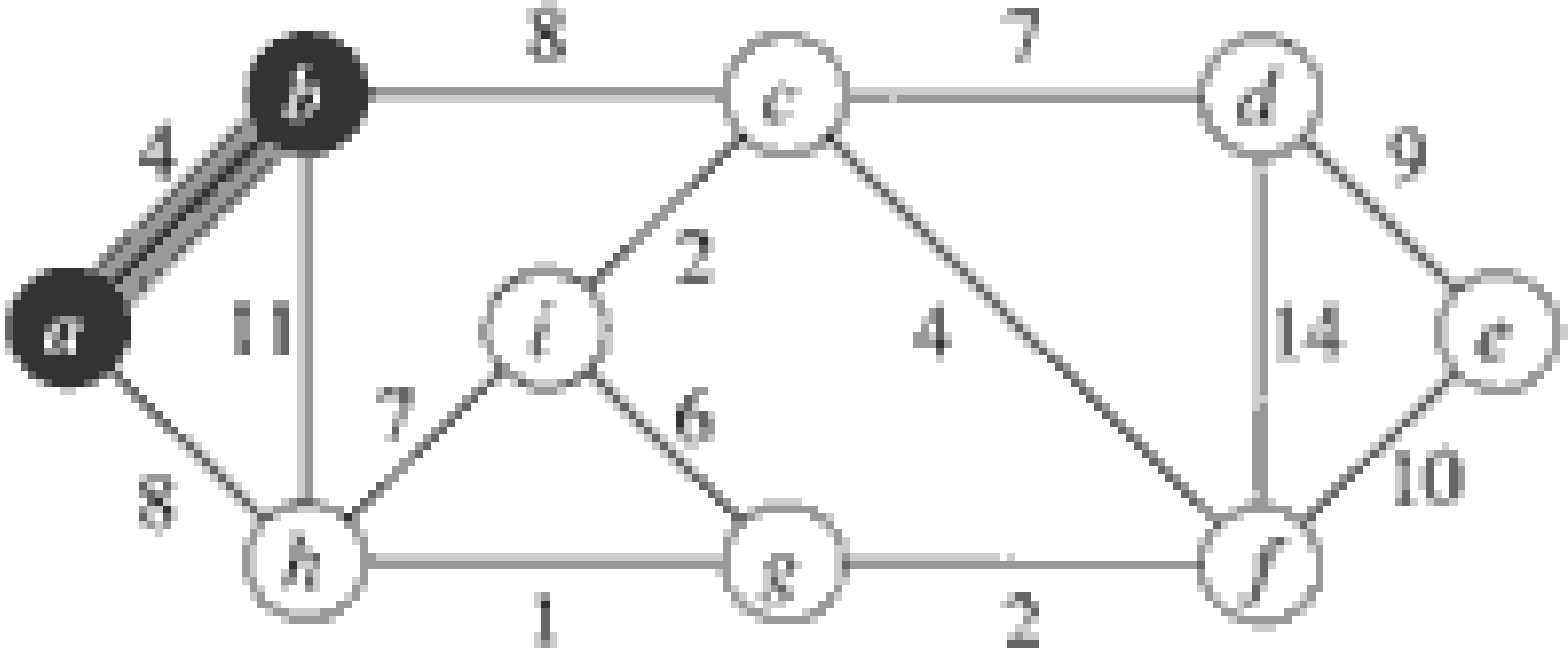


[Edges of A are shaded.]

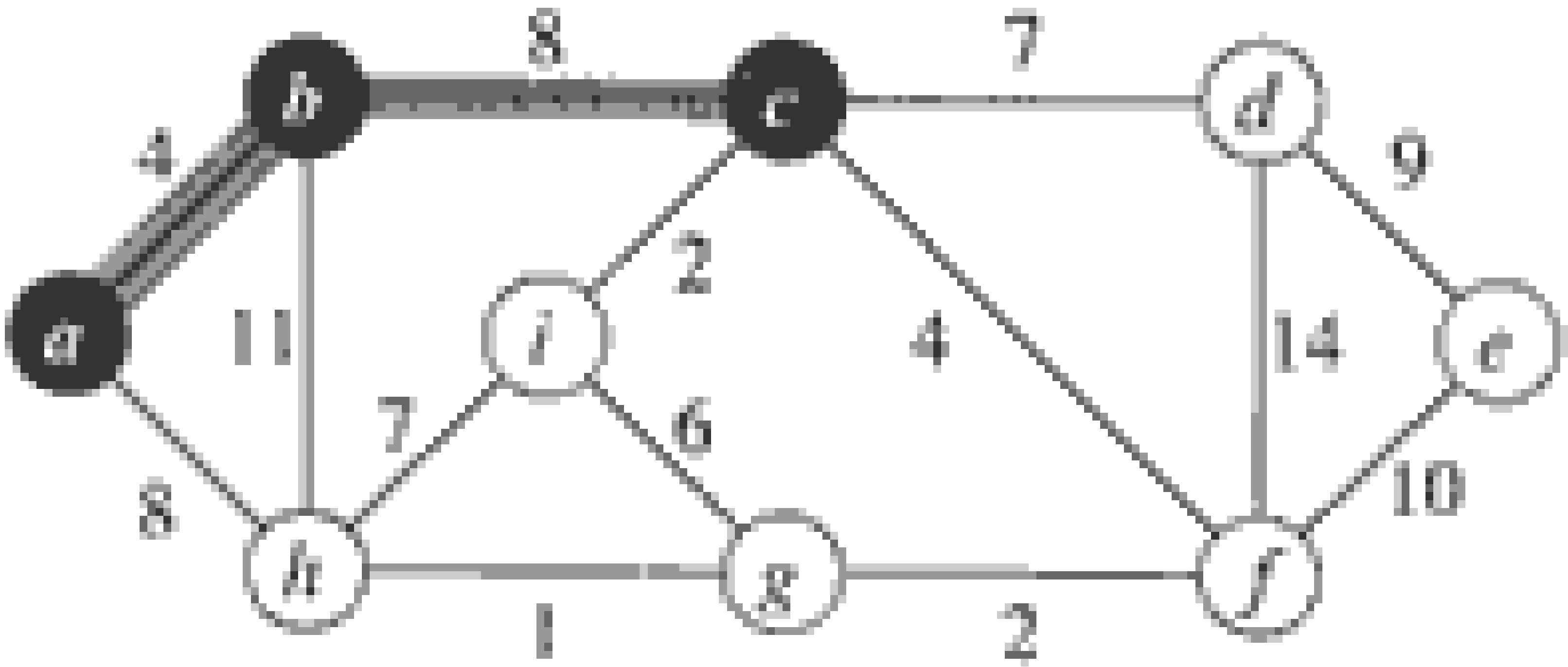
Prim’s Algorithm: Example



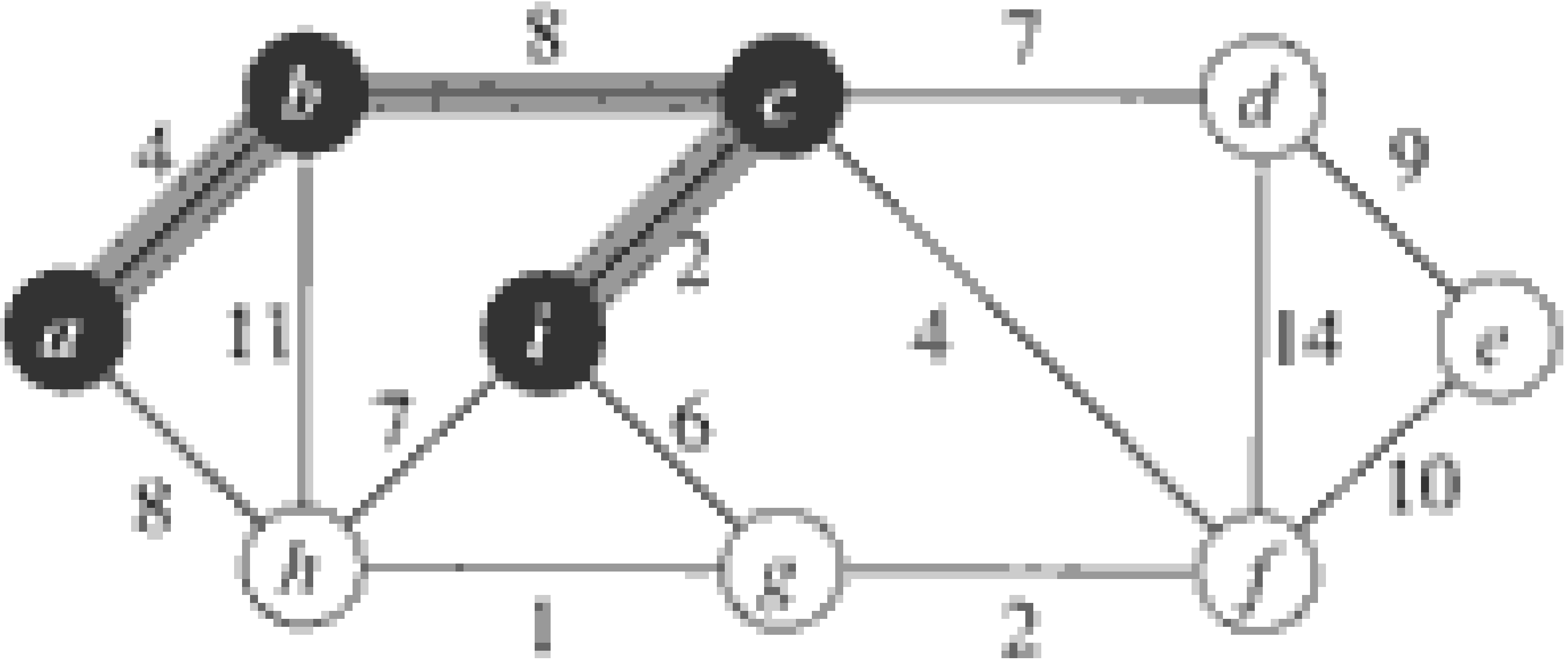
Prim’s Algorithm: Example



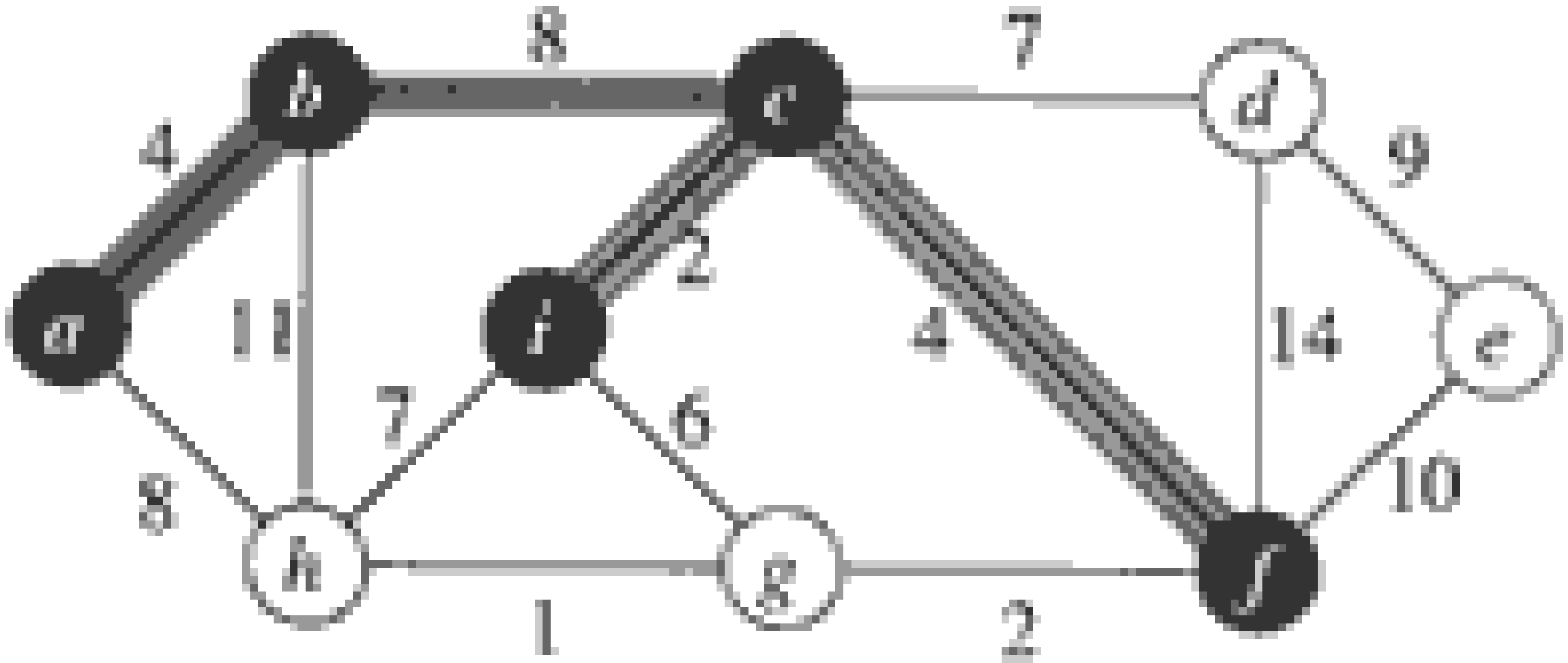
Prim’s Algorithm: Example



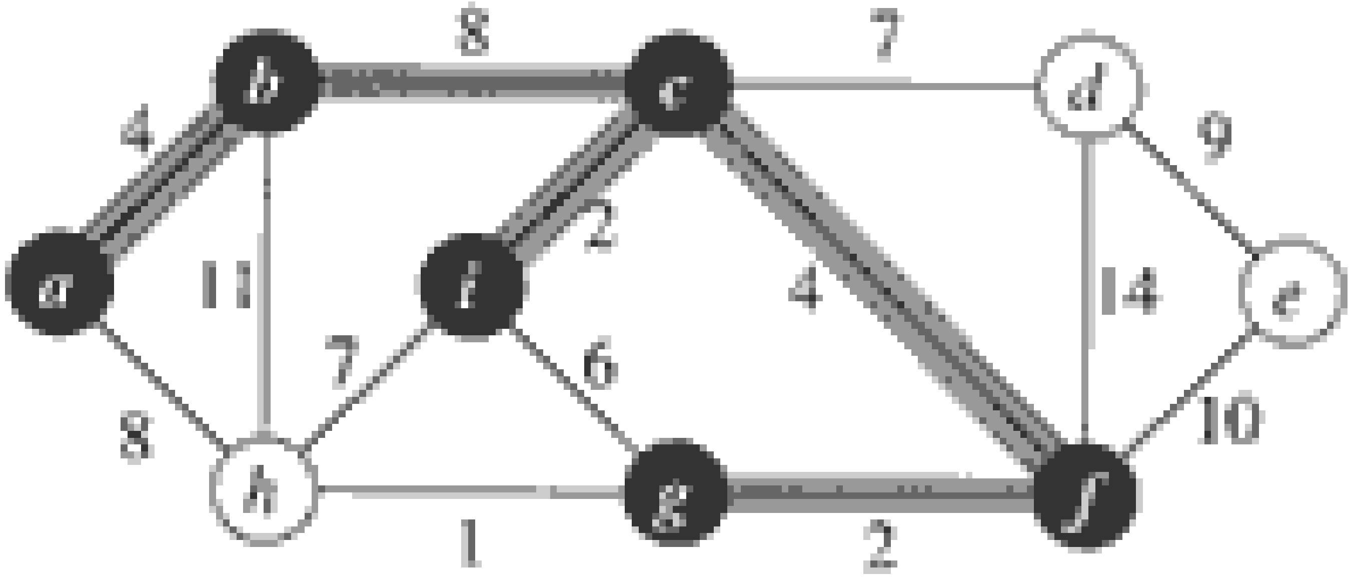
Prim’s Algorithm: Example



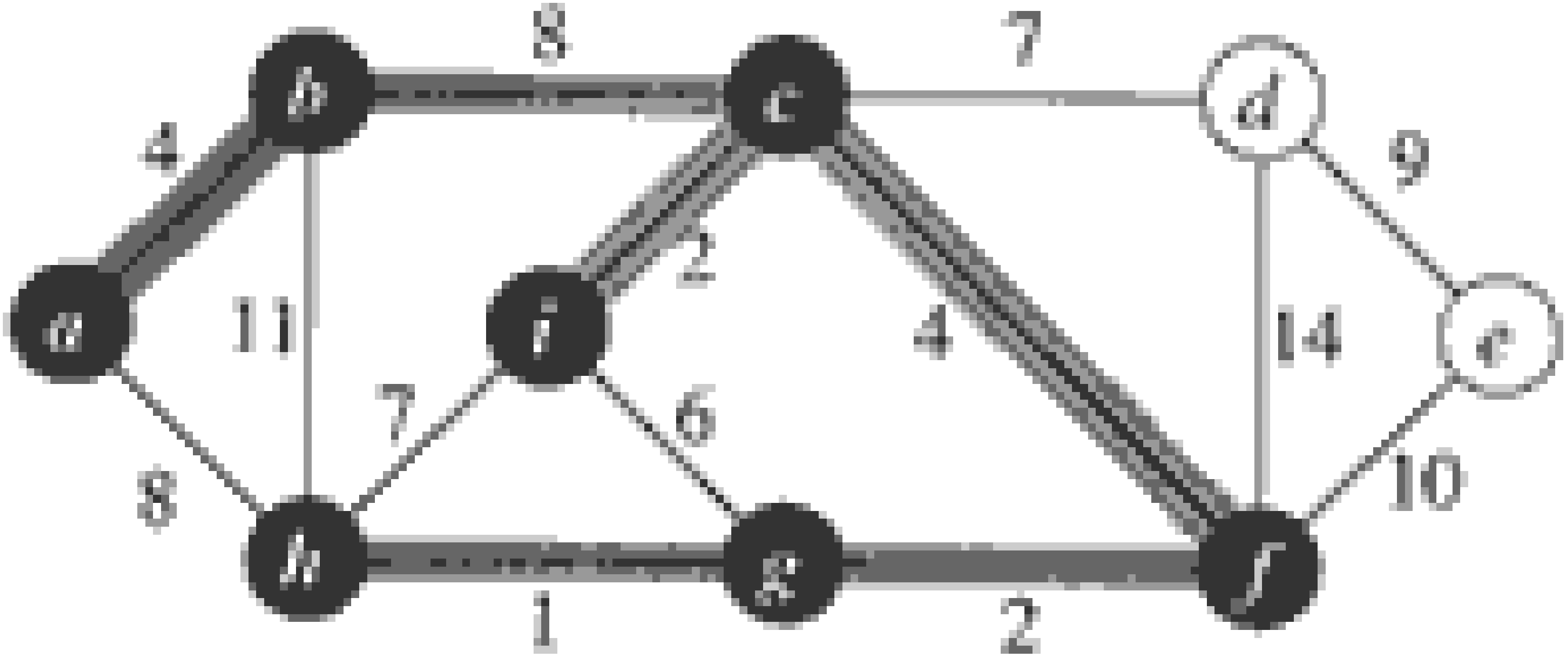
Prim's Algorithm: Example



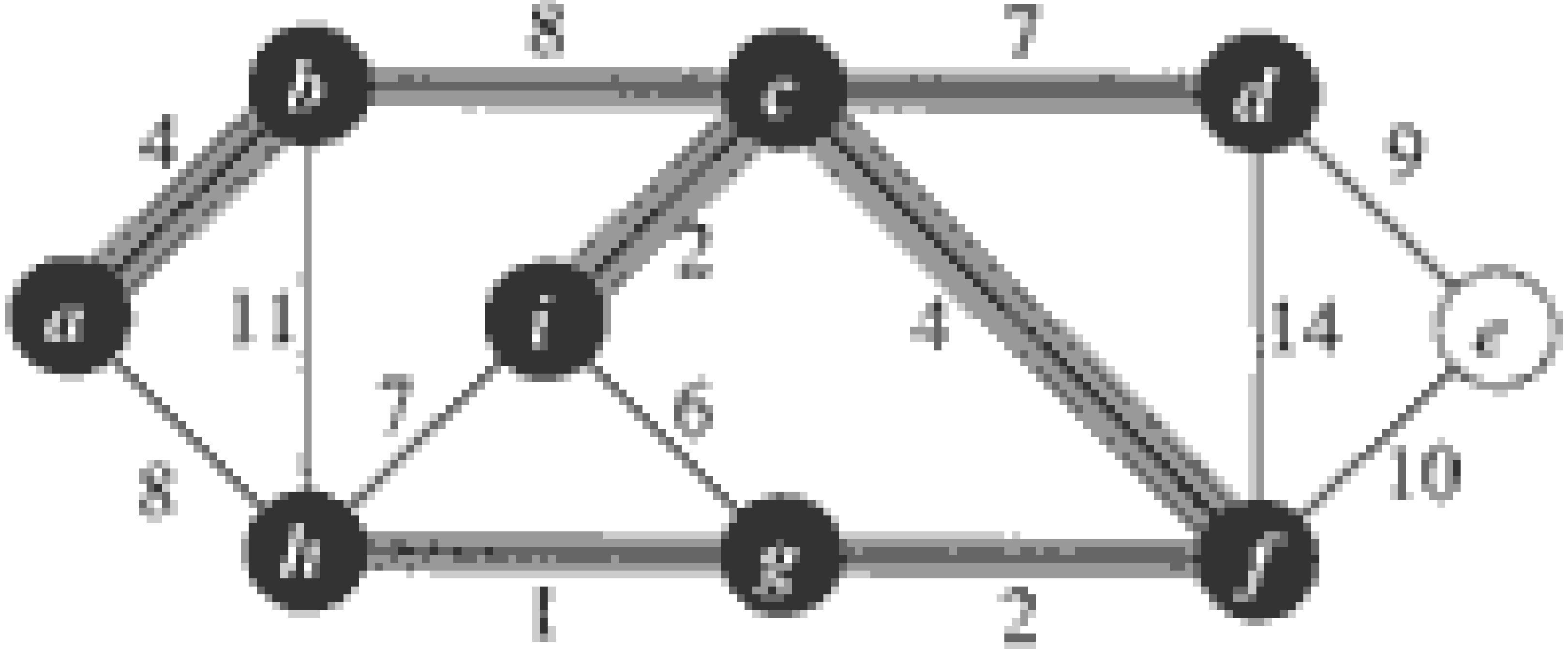
Prim's Algorithm: Example



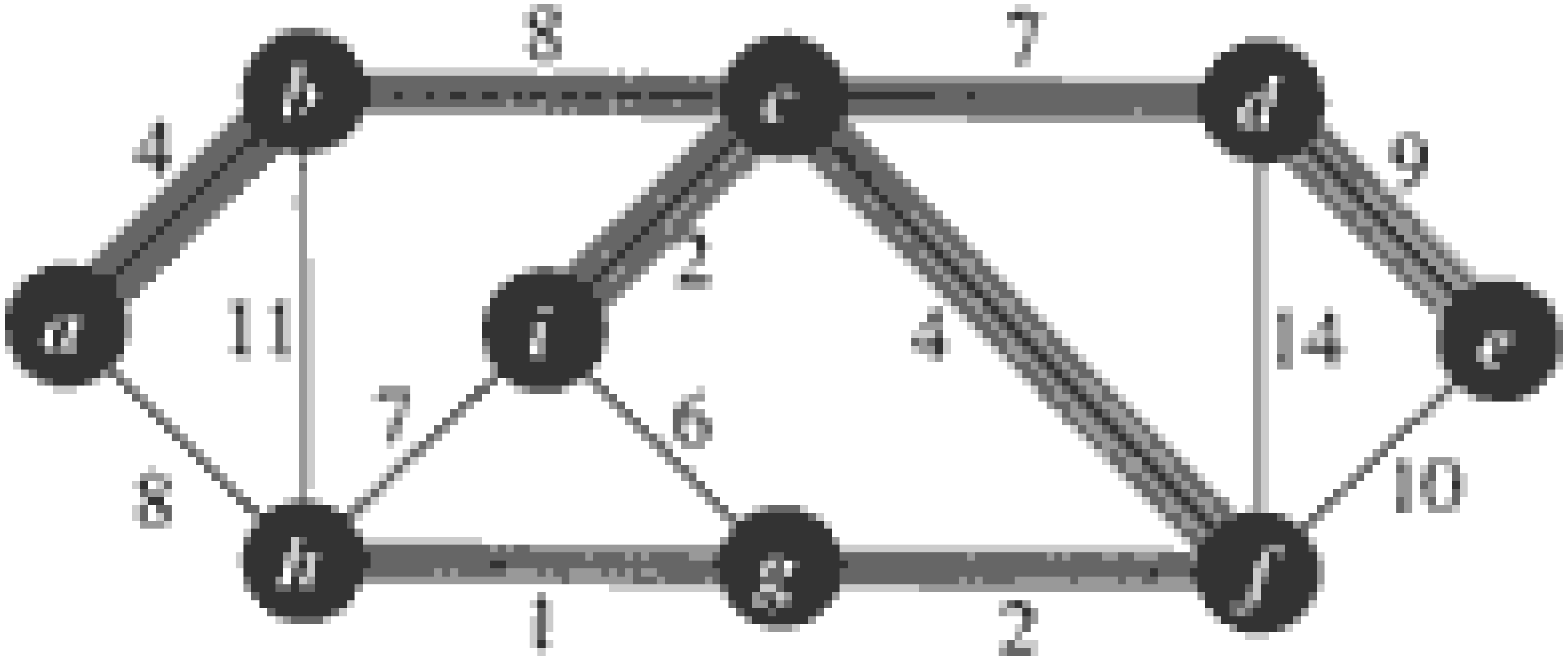
Prim’s Algorithm: Example



Prim’s Algorithm: Example



Prim’s Algorithm: Example



Finished!

Finding light edges quickly

- All vertices not in the partial MST formed by A reside in a min-priority queue.
- Key(v) is minimum weight of any edge (u,v), u$\in$ VA.
- Priority queue can be implemented as a min heap on key(v).
- Each vertex in queue knows its potential parent in partial MST by π[v].

Prim’s Algorithm

Let $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

Let $V_A = V - Q$

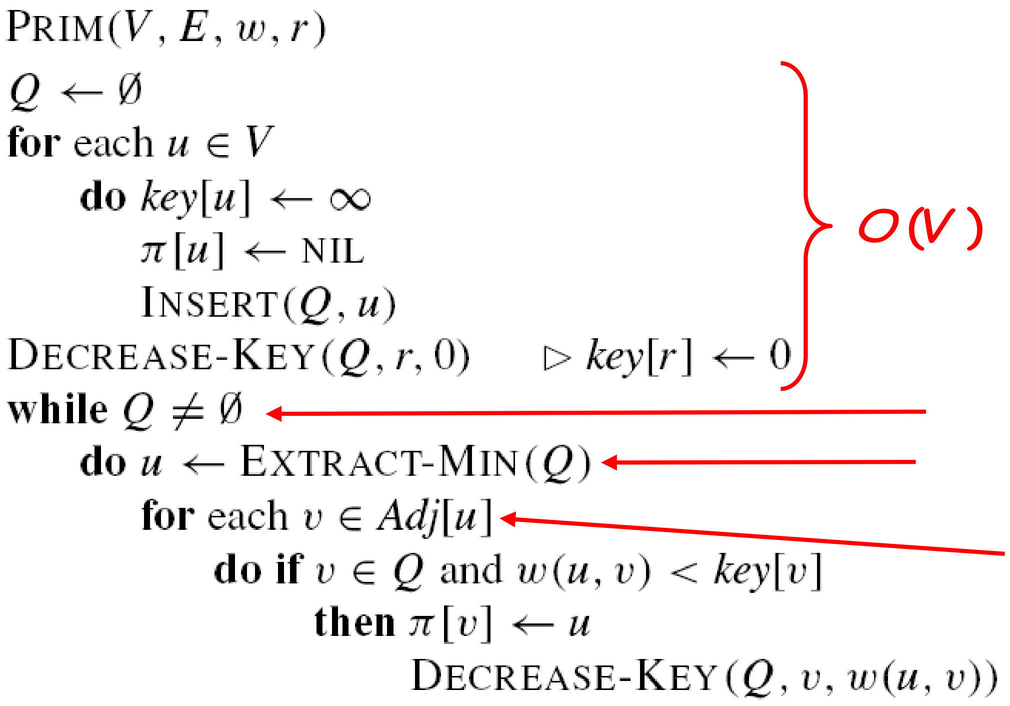
<loop-invariant>:

1. $\exists$ MST $T : A \in T$
2. $\forall$v $\in$ Q, if $\pi[v] \neq$ NIL

   then $key[v] =$ weight of light edge connecting v to V_A

```
PRIM$(V, E, w, r)$
$Q \leftarrow \emptyset$
for each $u \in V$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
       INSERT$(Q, u)$
DECREASE-KEY$(Q, r, 0)$        $\triangleright$ $key[r] \leftarrow 0$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                 then $\pi[v] \leftarrow u$
                      DECREASE-KEY$(Q, v, w(u, v))$
```

Prim’s Algorithm



```
PRIM($V, E, w, r$)
$Q \leftarrow \emptyset$
for each $u \in V$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
       INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)    $\triangleright$ $key[r] \leftarrow 0$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                 then $\pi[v] \leftarrow u$
                      DECREASE-KEY($Q, v, w(u, v)$)
```

$O(V)$

Executed $|V|$ times

$O(\log V)$

Executed $|E|$ times

$O(\log V)$

Running Time = $O(E \log V)$

Algorithm Comparison

Both Kruskal's and Prim's algorithm are greedy.

Kruskal's: Queue is static (constructed before loop)

Prim's: Queue is dynamic (keys adjusted as edges are encountered)

Thank You!