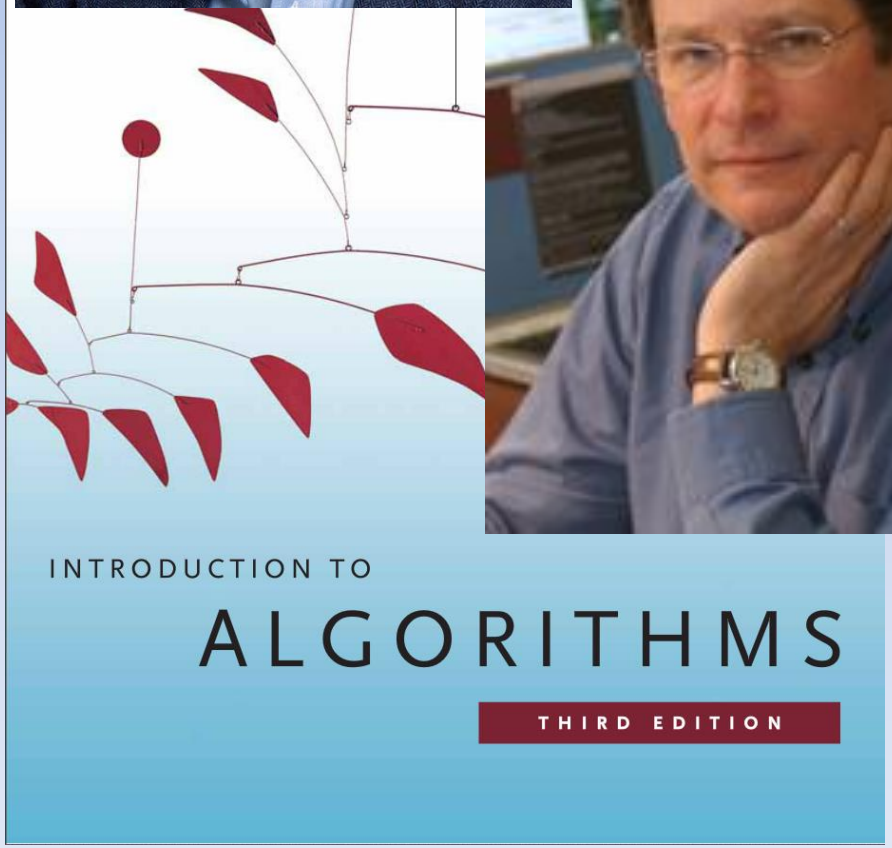
Design and Analysis of Algorithms

Course Review



Final

- Wednesday, December 13
- 11:00 – 1:00p
- McKee Fisk 208 (Classroom)

Final

- Chapters 1, 2, 3, 33, 4* (Master Method)
- Chapters 5, 10, 11, 12, 13

- Chapters 22* (dfs/bfs), 23* (primm/kruskal)
- Chapters 24, 25, 26

- Chapters 15* (LCS), 16* (Huffman), 17*

- Chapter 34*, 35

*Key Concepts

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *11*

2 Getting Started *16*
- 2.1 Insertion sort *16*
- 2.2 Analyzing algorithms *23*
- 2.3 Designing algorithms *29*

3 Growth of Functions *43*
- 3.1 Asymptotic notation *43*
- 3.2 Standard notations and common functions *53*

4 Divide-and-Conquer *65*
- 4.1 The maximum-subarray problem *68*
- 4.2 Strassen's algorithm for matrix multiplication *75*
- 4.3 The substitution method for solving recurrences *83*
- 4.4 The recursion-tree method for solving recurrences *88*
- 4.5 The master method for solving recurrences *93*
- ★ 4.6 Proof of the master theorem *97*

5 Probabilistic Analysis and Randomized Algorithms *114*
- 5.1 The hiring problem *114*
- 5.2 Indicator random variables *118*
- 5.3 Randomized algorithms *122*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *130*

31 Number-Theoretic Algorithms *926*
31.1 Elementary number-theoretic notions *927*
31.2 Greatest common divisor *933*
31.3 Modular arithmetic *939*
31.4 Solving modular linear equations *946*
31.5 The Chinese remainder theorem *950*
31.6 Powers of an element *954*
31.7 The RSA public-key cryptosystem *958*
★ 31.8 Primality testing *965*
★ 31.9 Integer factorization *975*

32 String Matching *985*
32.1 The naive string-matching algorithm *988*
32.2 The Rabin-Karp algorithm *990*
32.3 String matching with finite automata *995*
★ 32.4 The Knuth-Morris-Pratt algorithm *1002*

33 Computational Geometry *1014*
33.1 Line-segment properties *1015*
33.2 Determining whether any pair of segments intersects *1021*
33.3 Finding the convex hull *1029*
33.4 Finding the closest pair of points *1039*

34 NP-Completeness *1048*
34.1 Polynomial time *1053*
34.2 Polynomial-time verification *1061*
34.3 NP-completeness and reducibility *1067*
34.4 NP-completeness proofs *1078*
34.5 NP-complete problems *1086*

35 Approximation Algorithms *1106*
35.1 The vertex-cover problem *1108*
35.2 The traveling-salesman problem *1111*
35.3 The set-covering problem *1117*
35.4 Randomization and linear programming *1123*
35.5 The subset-sum problem *1128*

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *11*

2 Getting Started *16*
- 2.1 Insertion sort *16*
- 2.2 Analyzing algorithms *23*
- 2.3 Designing algorithms *29*

3 Growth of Functions *43*
- 3.1 Asymptotic notation *43*
- 3.2 Standard notations and common functions *53*

4 Divide-and-Conquer *65*
- 4.1 The maximum-subarray problem *68*
- 4.2 Strassen's algorithm for matrix multiplication *75*
- 4.3 The substitution method for solving recurrences *83*
- 4.4 The recursion-tree method for solving recurrences *88*
- 4.5 The master method for solving recurrences *93*
- ★ 4.6 Proof of the master theorem *97*

5 Probabilistic Analysis and Randomized Algorithms *114*
- 5.1 The hiring problem *114*
- 5.2 Indicator random variables *118*
- 5.3 Randomized algorithms *122*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *130*

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *11*

2 Getting Started *16*
- 2.1 Insertion sort *16*
- 2.2 Analyzing algorithms *23*
- 2.3 Designing algorithms *29*

3 Growth of Functions *43*
- 3.1 Asymptotic notation *43*
- 3.2 Standard notations and common functions *53*

4 Divide-and-Conquer *65*
- 4.1 The maximum-subarray problem *68*
- 4.2 Strassen's algorithm for matrix multiplication *75*
- 4.3 The substitution method for solving recurrences *83*
- 4.4 The recursion-tree method for solving recurrences *88*
- 4.5 The master method for solving recurrences *93*
- ★ 4.6 Proof of the master theorem *97*

5 Probabilistic Analysis and Randomized Algorithms *114*
- 5.1 The hiring problem *114*
- 5.2 Indicator random variables *118*
- 5.3 Randomized algorithms *122*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *130*

III Data Structures

Introduction *229*

10 Elementary Data Structures *232*
10.1 Stacks and queues *232*
10.2 Linked lists *236*
10.3 Implementing pointers and objects *241*
10.4 Representing rooted trees *246*

11 Hash Tables *253*
11.1 Direct-address tables *254*
11.2 Hash tables *256*
11.3 Hash functions *262*
11.4 Open addressing *269*
★ 11.5 Perfect hashing *277*

12 Binary Search Trees *286*
12.1 What is a binary search tree? *286*
12.2 Querying a binary search tree *289*
12.3 Insertion and deletion *294*
★ 12.4 Randomly built binary search trees *299*

13 Red-Black Trees *308*
13.1 Properties of red-black trees *308*
13.2 Rotations *312*
13.3 Insertion *315*
13.4 Deletion *323*

14 Augmenting Data Structures *339*
14.1 Dynamic order statistics *339*
14.2 How to augment a data structure *345*
14.3 Interval trees *348*

III Data Structures

Introduction *229*

10 Elementary Data Structures *232*
- 10.1 Stacks and queues *232*
- 10.2 Linked lists *236*
- 10.3 Implementing pointers and objects *241*
- 10.4 Representing rooted trees *246*

11 Hash Tables *253*
- 11.1 Direct-address tables *254*
- 11.2 Hash tables *256*
- 11.3 Hash functions *262*
- 11.4 Open addressing *269*
- ★ 11.5 Perfect hashing *277*

12 Binary Search Trees *286*
- 12.1 What is a binary search tree? *286*
- 12.2 Querying a binary search tree *289*
- 12.3 Insertion and deletion *294*
- ★ 12.4 Randomly built binary search trees *299*

13 Red-Black Trees *308*
- 13.1 Properties of red-black trees *308*
- 13.2 Rotations *312*
- 13.3 Insertion *315*
- 13.4 Deletion *323*

14 Augmenting Data Structures *339*
- 14.1 Dynamic order statistics *339*
- 14.2 How to augment a data structure *345*
- 14.3 Interval trees *348*

VI Graph Algorithms

Introduction *587*

22 Elementary Graph Algorithms *589*
- 22.1 Representations of graphs *589*
- 22.2 Breadth-first search *594*
- 22.3 Depth-first search *603*
- 22.4 Topological sort *612*
- 22.5 Strongly connected components *615*

23 Minimum Spanning Trees *624*
- 23.1 Growing a minimum spanning tree *625*
- 23.2 The algorithms of Kruskal and Prim *631*

24 Single-Source Shortest Paths *643*
- 24.1 The Bellman-Ford algorithm *651*
- 24.2 Single-source shortest paths in directed acyclic graphs *655*
- 24.3 Dijkstra's algorithm *658*
- 24.4 Difference constraints and shortest paths *664*
- 24.5 Proofs of shortest-paths properties *671*

25 All-Pairs Shortest Paths *684*
- 25.1 Shortest paths and matrix multiplication *686*
- 25.2 The Floyd-Warshall algorithm *693*
- 25.3 Johnson's algorithm for sparse graphs *700*

26 Maximum Flow *708*
- 26.1 Flow networks *709*
- 26.2 The Ford-Fulkerson method *714*
- 26.3 Maximum bipartite matching *732*
- ★ 26.4 Push-relabel algorithms *736*
- ★ 26.5 The relabel-to-front algorithm *748*

VI Graph Algorithms

Introduction *587*

22 Elementary Graph Algorithms *589*
- 22.1 Representations of graphs *589*
- 22.2 Breadth-first search *594*
- 22.3 Depth-first search *603*
- 22.4 Topological sort *612*
- 22.5 Strongly connected components *615*

23 Minimum Spanning Trees *624*
- 23.1 Growing a minimum spanning tree *625*
- 23.2 The algorithms of Kruskal and Prim *631*

24 Single-Source Shortest Paths *643*
- 24.1 The Bellman-Ford algorithm *651*
- 24.2 Single-source shortest paths in directed acyclic graphs *655*
- 24.3 Dijkstra's algorithm *658*
- 24.4 Difference constraints and shortest paths *664*
- 24.5 Proofs of shortest-paths properties *671*

25 All-Pairs Shortest Paths *684*
- 25.1 Shortest paths and matrix multiplication *686*
- 25.2 The Floyd-Warshall algorithm *693*
- 25.3 Johnson's algorithm for sparse graphs *700*

26 Maximum Flow *708*
- 26.1 Flow networks *709*
- 26.2 The Ford-Fulkerson method *714*
- 26.3 Maximum bipartite matching *732*
- ★ 26.4 Push-relabel algorithms *736*
- ★ 26.5 The relabel-to-front algorithm *748*

IV Advanced Design and Analysis Techniques

Introduction *357*

15 Dynamic Programming *359*

15.1 Rod cutting *360*
15.2 Matrix-chain multiplication *370*
15.3 Elements of dynamic programming *378*
15.4 Longest common subsequence *390*
15.5 Optimal binary search trees *397*

16 Greedy Algorithms *414*

16.1 An activity-selection problem *415*
16.2 Elements of the greedy strategy *423*
16.3 Huffman codes *428*
★ 16.4 Matroids and greedy methods *437*
★ 16.5 A task-scheduling problem as a matroid *443*

17 Amortized Analysis *451*

17.1 Aggregate analysis *452*
17.2 The accounting method *456*
17.3 The potential method *459*
17.4 Dynamic tables *463*

IV Advanced Design and Analysis Techniques

Introduction *357*

15 Dynamic Programming *359*

15.1 Rod cutting *360*
15.2 Matrix-chain multiplication *370*
15.3 Elements of dynamic programming *378*
15.4 Longest common subsequence *390*
15.5 Optimal binary search trees *397*

16 Greedy Algorithms *414*

16.1 An activity-selection problem *415*
16.2 Elements of the greedy strategy *423*
16.3 Huffman codes *428*
★ 16.4 Matroids and greedy methods *437*
★ 16.5 A task-scheduling problem as a matroid *443*

17 Amortized Analysis *451*

17.1 Aggregate analysis *452*
17.2 The accounting method *456*
17.3 The potential method *459*
17.4 Dynamic tables *463*

IV Advanced Design and Analysis Techniques

Introduction *357*

15 Dynamic Programming *359*

15.1 Rod cutting *360*
15.2 Matrix-chain multiplication *370*
15.3 Elements of dynamic programming *378*
15.4 Longest common subsequence *390*
15.5 Optimal binary search trees *397*

16 Greedy Algorithms *414*

16.1 An activity-selection problem *415*
16.2 Elements of the greedy strategy *423*
16.3 Huffman codes *428*
★ 16.4 Matroids and greedy methods *437*
★ 16.5 A task-scheduling problem as a matroid *443*

17 Amortized Analysis *451*

17.1 Aggregate analysis *452*
17.2 The accounting method *456*
17.3 The potential method *459*
17.4 Dynamic tables *463*

Selected Topics

31 Number-Theoretic Algorithms *926*
31.1 Elementary number-theoretic notions *927*
31.2 Greatest common divisor *933*
31.3 Modular arithmetic *939*
31.4 Solving modular linear equations *946*
31.5 The Chinese remainder theorem *950*
31.6 Powers of an element *954*
31.7 The RSA public-key cryptosystem *958*
★ 31.8 Primality testing *965*
★ 31.9 Integer factorization *975*

32 String Matching *985*
32.1 The naive string-matching algorithm *988*
32.2 The Rabin-Karp algorithm *990*
32.3 String matching with finite automata *995*
★ 32.4 The Knuth-Morris-Pratt algorithm *1002*

33 Computational Geometry *1014*
33.1 Line-segment properties *1015*
33.2 Determining whether any pair of segments intersects *1021*
33.3 Finding the convex hull *1029*
33.4 Finding the closest pair of points *1039*

34 NP-Completeness *1048*
34.1 Polynomial time *1053*
34.2 Polynomial-time verification *1061*
34.3 NP-completeness and reducibility *1067*
34.4 NP-completeness proofs *1078*
34.5 NP-complete problems *1086*

35 Approximation Algorithms *1106*
35.1 The vertex-cover problem *1108*
35.2 The traveling-salesman problem *1111*
35.3 The set-covering problem *1117*
35.4 Randomization and linear programming *1123*
35.5 The subset-sum problem *1128*

Selected Topics

31 Number-Theoretic Algorithms *926*
31.1 Elementary number-theoretic notions *927*
31.2 Greatest common divisor *933*
31.3 Modular arithmetic *939*
31.4 Solving modular linear equations *946*
31.5 The Chinese remainder theorem *950*
31.6 Powers of an element *954*
31.7 The RSA public-key cryptosystem *958*
★ 31.8 Primality testing *965*
★ 31.9 Integer factorization *975*

32 String Matching *985*
32.1 The naive string-matching algorithm *988*
32.2 The Rabin-Karp algorithm *990*
32.3 String matching with finite automata *995*
★ 32.4 The Knuth-Morris-Pratt algorithm *1002*

33 Computational Geometry *1014*
33.1 Line-segment properties *1015*
33.2 Determining whether any pair of segments intersects *1021*
33.3 Finding the convex hull *1029*
33.4 Finding the closest pair of points *1039*

34 NP-Completeness *1048*
34.1 Polynomial time *1053*
34.2 Polynomial-time verification *1061*
34.3 NP-completeness and reducibility *1067*
34.4 NP-completeness proofs *1078*
34.5 NP-complete problems *1086*

35 Approximation Algorithms *1106*
35.1 The vertex-cover problem *1108*
35.2 The traveling-salesman problem *1111*
35.3 The set-covering problem *1117*
35.4 Randomization and linear programming *1123*
35.5 The subset-sum problem *1128*

Final

- Chapters 1, 2, 3, 33, 4* (Master Method)
- Chapters 5, 10, 11, 12, 13, 18

- Chapters 22* (dfs/bfs), 23* (primm/kruskal)

- Chapters 15* (LCS), 16* (Huffman), 17*

- Chapter 34*, 35

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *11*

2 Getting Started *16*
- 2.1 Insertion sort *16*
- 2.2 Analyzing algorithms *23*
- 2.3 Designing algorithms *29*

3 Growth of Functions *43*
- 3.1 Asymptotic notation *43*
- 3.2 Standard notations and common functions *53*

4 Divide-and-Conquer *65*
- 4.1 The maximum-subarray problem *68*
- 4.2 Strassen's algorithm for matrix multiplication *75*
- 4.3 The substitution method for solving recurrences *83*
- 4.4 The recursion-tree method for solving recurrences *88*
- 4.5 The master method for solving recurrences *93*
- ★ 4.6 Proof of the master theorem *97*

5 Probabilistic Analysis and Randomized Algorithms *114*
- 5.1 The hiring problem *114*
- 5.2 Indicator random variables *118*
- 5.3 Randomized algorithms *122*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *130*

Topic 1: Master Method

- Use the Master Method to solve the recurrences below?
  - $T(N) = 3*T(N/3) + 7$
  - $T(N) = 4*T(N/4) + N$
  - $T(N) = 3*T(N/4) + 7$
  - $T(N) = 4*T(N/3) + N^2$
  - $T(N) = 4*T(N/3) + 5$

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *11*

2 Getting Started *16*
- 2.1 Insertion sort *16*
- 2.2 Analyzing algorithms *23*
- 2.3 Designing algorithms *29*

3 Growth of Functions *43*
- 3.1 Asymptotic notation *43*
- 3.2 Standard notations and common functions *53*

4 Divide-and-Conquer *65*
- 4.1 The maximum-subarray problem *68*
- 4.2 Strassen's algorithm for matrix multiplication *75*
- 4.3 The substitution method for solving recurrences *83*
- 4.4 The recursion-tree method for solving recurrences *88*
- 4.5 The master method for solving recurrences *93*
- ★ 4.6 Proof of the master theorem *97*

5 Probabilistic Analysis and Randomized Algorithms *114*
- 5.1 The hiring problem *114*
- 5.2 Indicator random variables *118*
- 5.3 Randomized algorithms *122*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *130*

Divide-and-Conquer

- **Divide** problem into some number of smaller instances of the same problem.
- **Conquer** the subproblems recursively until small enough to just solve.
- **Combine** the subproblem solutions into final solution.

Recurrences w/ Divide & Conquer

- Recurrences define functions recursively
- Recurrence describes function in terms of value on smaller inputs.

Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two equal parts
  - Recursive solves each of the parts
  - Combines the two sub-solutions into the final solution doing constant work to combine solutions.
    - Comparing the two max's
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(1), & if\ n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two equal parts
  - Recursive solves each of the parts
  - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

Recurrence Example

- Assume we have an algorithm that:
  - breaks a problem into two **unequal** parts
    - First part is 2/3 of items
    - Second part is 1/3 of items
  - Combines the two sub-solutions into the final solution doing work proportional to input size.
- Model this situation with the recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

Divide & Conquer Summary

- Split the problem in half and recursively solve smaller versions
  - Creating a Lg(n) recursion tree
- Merge sub solutions in $\boldsymbol{\Theta}(n)$
  - Each level of recursion has the same n items
  - Depth of recursion is $\boldsymbol{Lg\ n}$
- Smart Merge Procedure needed!

Divide & Conquer w/ Merge sort

- Sort Problem: Take an n-element sequence of numbers and find a permutation where elements are ordered smallest to largest.
- Merge sort:
  - **Divide** the n-element sequence into two subsequence of n/2 elements each.
  - **Conquer** the two subsequences recursively.
  - **Combine** the two sorted subsequences for the final solution.

Divide & Conquer w/ Merge sort

- Recursion bottoms out when the sequence is of size 1, which is sorted by definition.
- Key operation is MERGING two sorted sequences.
- Merging is done in time $\boldsymbol{\Theta}(n)$ where n is the total number of items to be merged.

Merge

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Copy to Temp

Loop through N Items

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Develop Recurrence Relation
  - Divide step (and base case): $\boldsymbol{\Theta}(1)$
  - Conquer: T(n) = T(n/2) + T(n/2)
  - Combine: Merge procedure is $\boldsymbol{\Theta}(n)$

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Model final recurrence:

$$T(n) = \begin{cases} \boldsymbol{\Theta}(1), & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n), & if\ n > 1 \end{cases}$$

- **T(n) =** $\boldsymbol{\Theta}(n\ Lg\ n)$

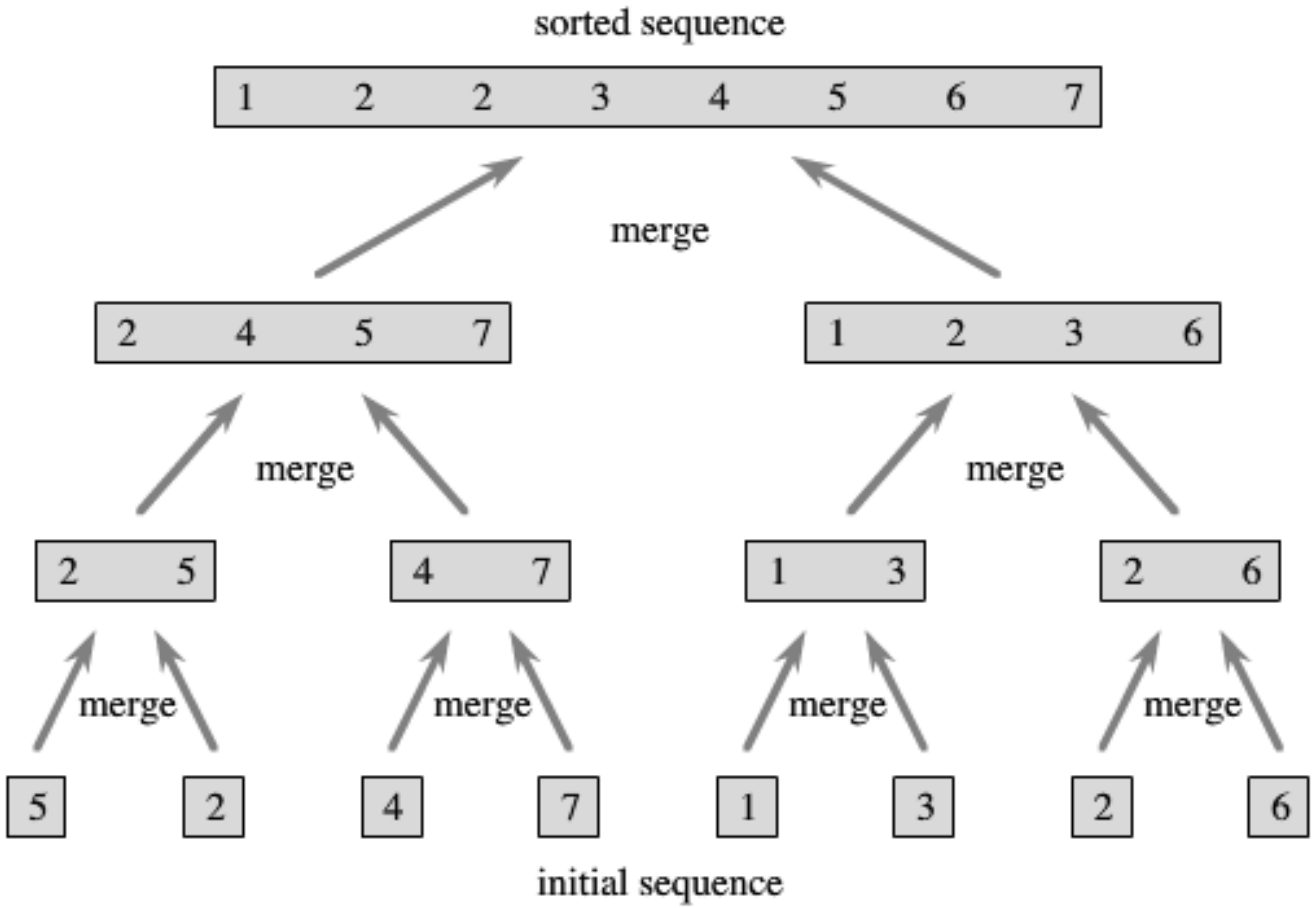
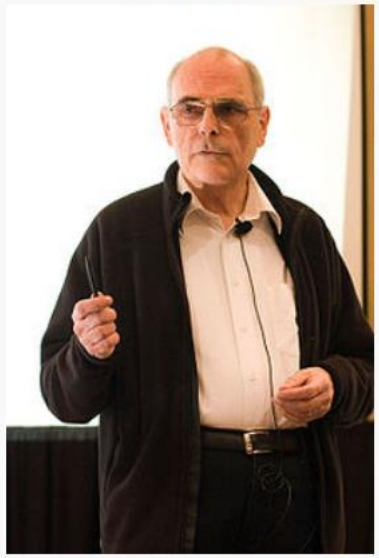


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Strassen Matrix Multiplication

- First published in 1969
- Improves upon the standard matrix multiplication algorithm $O(n^3)$
- $O(n^{\lg 7}) = n^{2.807355}$

Volker Strassen



Volker Strassen giving the Knuth Prize lecture at SODA 2009

Born	April 29, 1936 (age 79) Düsseldorf-Gerresheim, Prussia
Nationality	German
Fields	Mathematics
Institutions	University of Konstanz
Alma mater	University of Göttingen
Doctoral advisor	Konrad Jacobs (de)
Doctoral students	Peter Bürgisser Joos Heintz Joachim von zur Gathen

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

- Triple loop: n^3

Simple Divide & Conquer
Break Matrix into 4 Quadrants

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?
- Base Case:

$$T(1) = \Theta(1) . \tag{4.15}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?
- Recursive Case:
  - Partition Matrices: $\boldsymbol{\Theta}(1)$ w/ Indices Calculations
  - Add two n/2 Square Matrices: $(n/2)^2$ elements each
    - $n^2/4 \Rightarrow \boldsymbol{\Theta}(n^2)$
  - 8 Recursive calls of size n/2

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) \,. \end{aligned} \tag{4.16}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases} \tag{4.17}$$

- We'll look closer at solving this recurrence, but for now:
  - T(n) = $\boldsymbol{\Theta}(n^3)$
- We have NOT done better!!
- Where's the problem??

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

- Have we done better?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases}$$

TOO MANY SUBPROBLEMS!!

- We'll look closer at solving this recurrence, but for now:
  - T(n) = $\boldsymbol{\Theta}(n^3)$
- We have NOT done better!!
- Where's the problem??
  - Strassen's Improvement… FEWER SUBPROBLEMS!!!

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Strassen's Method

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

SEVEN Matrix Products!!

- Key Contribution: 7 Recursive Calls!!!
  - NOT 8!!

Recurrence Relation w/ Strassen's Method

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \tag{4.18}$$

SEVEN Matrix Products!!

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices cre[illegible] in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. E[illegible] $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

SEVEN Matrix Products!!

Recurrence Relation w/ Strassen's Method

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases} \tag{4.18}$$

SEVEN Matrix Products!!

- Instead of T(n) = $\boldsymbol{\Theta}(n^3)$
  - $\boldsymbol{\Theta}\left(n^{\log_2 8}\right)$
- For Strassen's Algorithm $\boldsymbol{\Theta}\left(n^{\log_2 7}\right)$

Strassen's Method w/ 10 Matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

$$\begin{aligned}
S_1 &= B_{12} - B_{22}\,, \\
S_2 &= A_{11} + A_{12}\,, \\
S_3 &= A_{21} + A_{22}\,, \\
S_4 &= B_{21} - B_{11}\,, \\
S_5 &= A_{11} + A_{22}\,, \\
S_6 &= B_{11} + B_{22}\,, \\
S_7 &= A_{12} - A_{22}\,, \\
S_8 &= B_{21} + B_{22}\,, \\
S_9 &= A_{11} - A_{21}\,, \\
S_{10} &= B_{11} + B_{12}\,.
\end{aligned}$$

Strassen's Method w/ 10 Matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

7 Matrix Multiplies w/ 10 Matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \,, \\
S_2 &= A_{11} + A_{12} \,, \\
S_3 &= A_{21} + A_{22} \,, \\
S_4 &= B_{21} - B_{11} \,, \\
S_5 &= A_{11} + A_{22} \,, \\
S_6 &= B_{11} + B_{22} \,, \\
S_7 &= A_{12} - A_{22} \,, \\
S_8 &= B_{21} + B_{22} \,, \\
S_9 &= A_{11} - A_{21} \,, \\
S_{10} &= B_{11} + B_{12} \,.
\end{aligned}
$$

Strassen's Method w/ 10 Matrices (TRICKY)

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

7 Matrix Multiplies w/ 10 Matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \,, \\
S_2 &= A_{11} + A_{12} \,, \\
S_3 &= A_{21} + A_{22} \,, \\
S_4 &= B_{21} - B_{11} \,, \\
S_5 &= A_{11} + A_{22} \,, \\
S_6 &= B_{11} + B_{22} \,, \\
S_7 &= A_{12} - A_{22} \,, \\
S_8 &= B_{21} + B_{22} \,, \\
S_9 &= A_{11} - A_{21} \,, \\
S_{10} &= B_{11} + B_{12} \,.
\end{aligned}
$$

Strassen's Method w/ 7 Sub-Matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}
$$

$$C_{11} = P_5 + P_4 - P_2 + P_6 \,.$$

7 Matrix Multiplies w/ 10 Matrices

$$
\begin{array}{lllll}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} & & & & \\
 & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & \\
 & - A_{11} \cdot B_{22} & & & - A_{12} \cdot B_{22} \\
 & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & + A_{12} \cdot B_{21} \,,
\end{array}
$$

$$C_{12} = P_1 + P_2 ,$$

and so C_{12} equals

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad\qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad\qquad + A_{12} \cdot B_{22} , \end{array}$$

corresponding to equation (4.12).

Setting

$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad\qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad\qquad + A_{22} \cdot B_{21} , \end{array}$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 ,$$

so that C_{22} equals

$$\begin{array}{l} A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\ \qquad\qquad - A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad + A_{11} \cdot B_{12} \\ \qquad\qquad\qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad\qquad\qquad - A_{21} \cdot B_{11} \\ - A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12} \\ \hline \qquad\qquad\qquad\qquad A_{22} \cdot B_{22} \qquad\qquad\qquad\qquad + A_{21} \cdot B_{12} , \end{array}$$

Solving Recurrences

Guess & Prove

- $T(N) = 2T(N/2) + O(N)$
- GUESS: $O(N\lg N)$

- Prove w/ Induction
  1. Prove a base case
  2. Prove $T(K)$ w/ assuming $T(k-1)$

Good Guesses

- Unfortunately, Good Guesses are Difficult!
- No general way to guess!

- If recurrence is similar to one seen before, guessing similar solution reasonable!

Quick Question: 4.3-1

- Show $T(N) = T(N-1)+N$ is $O(N^2)$

Quick Question: 4.3-1

- Show T(N) = T(N-1)+N is $O(N^2)$

- Guessing $T(N) \leq cn^2$ for some constant c>0

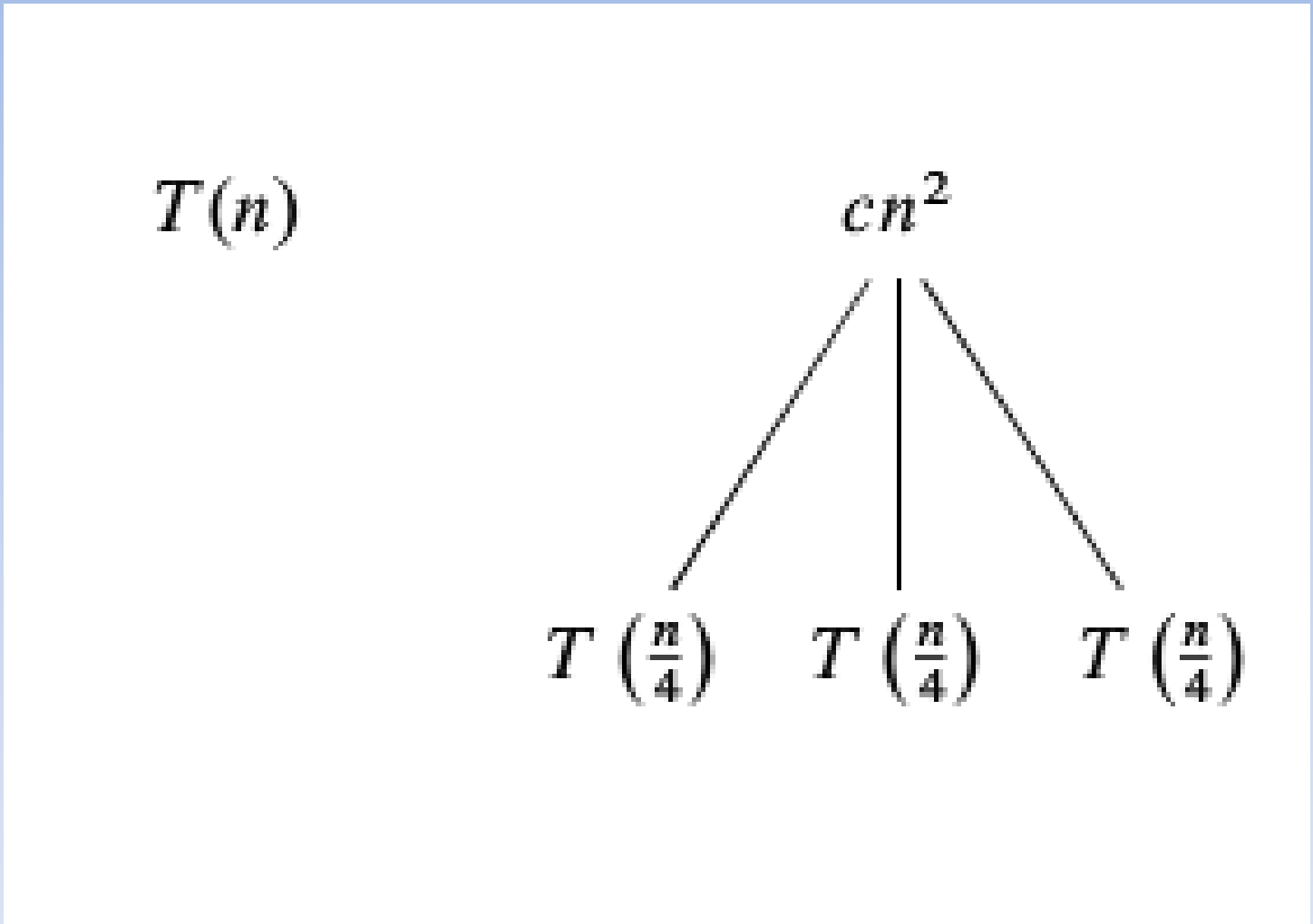
$$
\begin{aligned}
T(N) &= T(N-1) + N \\
&\leq c(N-1)^2 + N \\
&\leq c(N^2 - 2N + 1) + N \\
&\leq cN^2 - 2cN + c + N \\
&\leq cN^2 - c(1-2N) + N
\end{aligned}
$$

- $cN^2 - c(1-2N) + N \leq cN^2$
  - w/ $c \geq N/(2N-1)$
- $T(1) = 1 \leq c1^2$
- SO:
  - $n_0 = 1$
  - c = 1

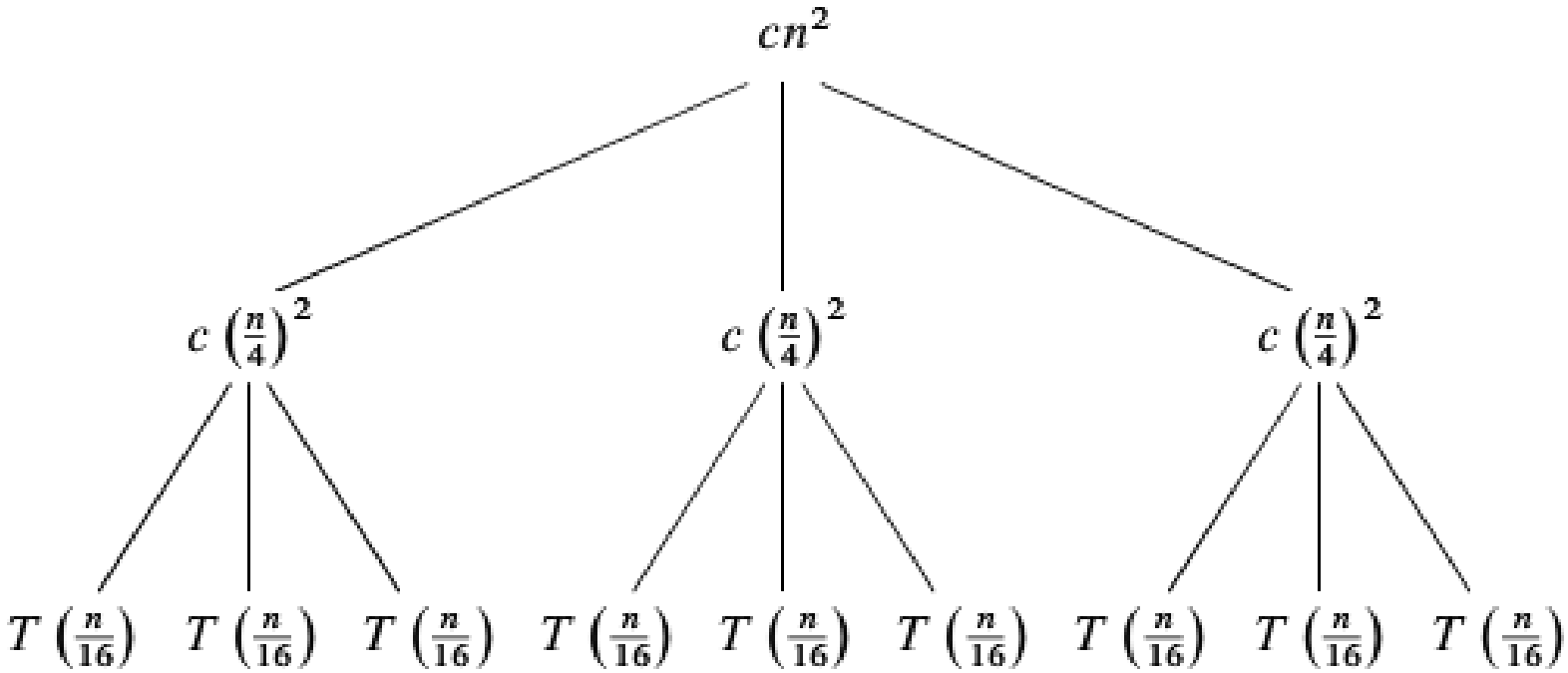
Solving Recurrences w/ Recursion-Tree Method

- Drawing the recursion tree helps to understand the recursive relations
- Recursion tree helps devise a good guess!
- Each node represents the cost of a single subproblem somewhere in the set of recursive invocations.
- Sum costs within each level of tree
- Count the number of levels in tree

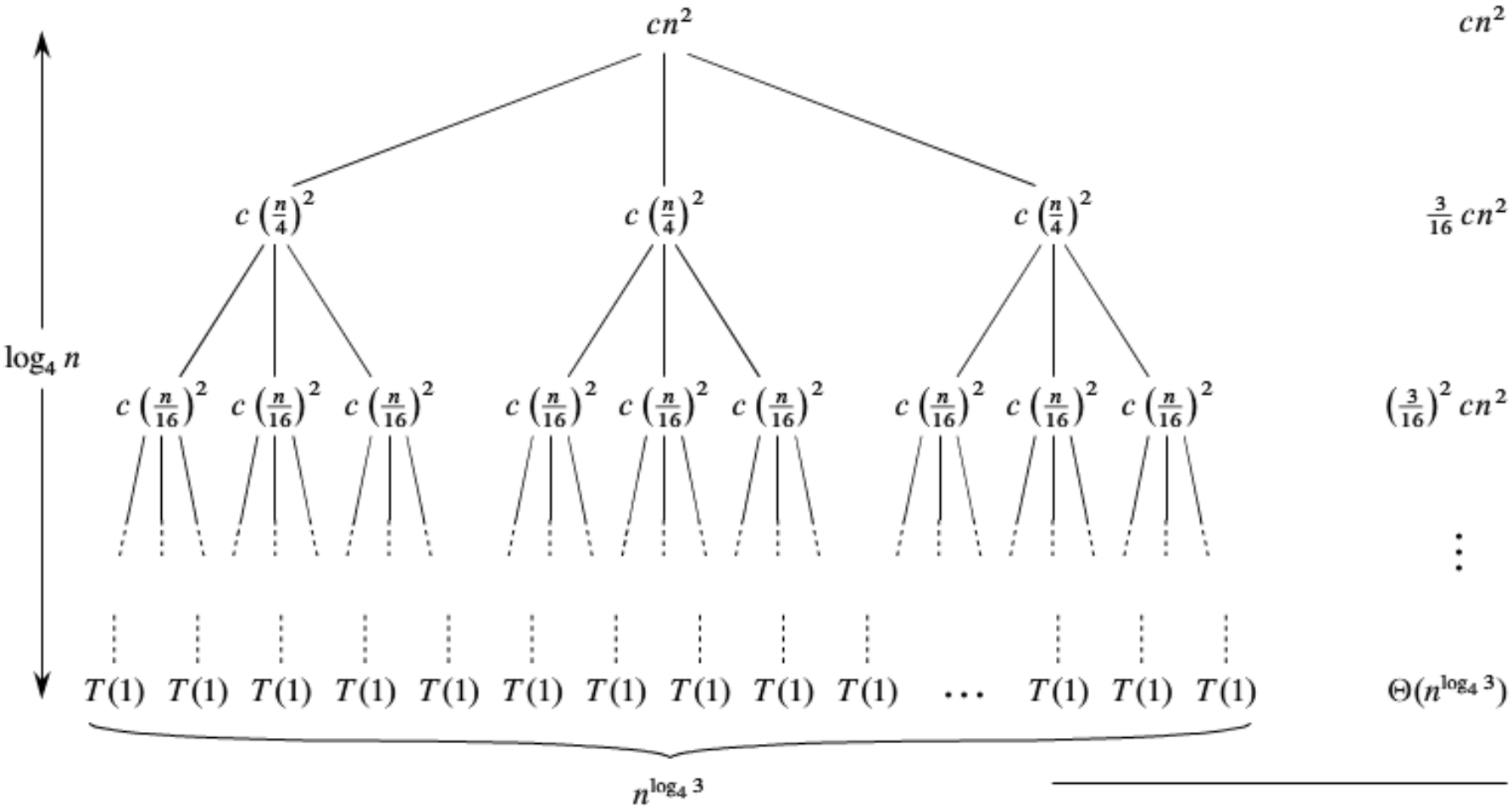
$T(N) = 3T(N/4) + n^2$


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$

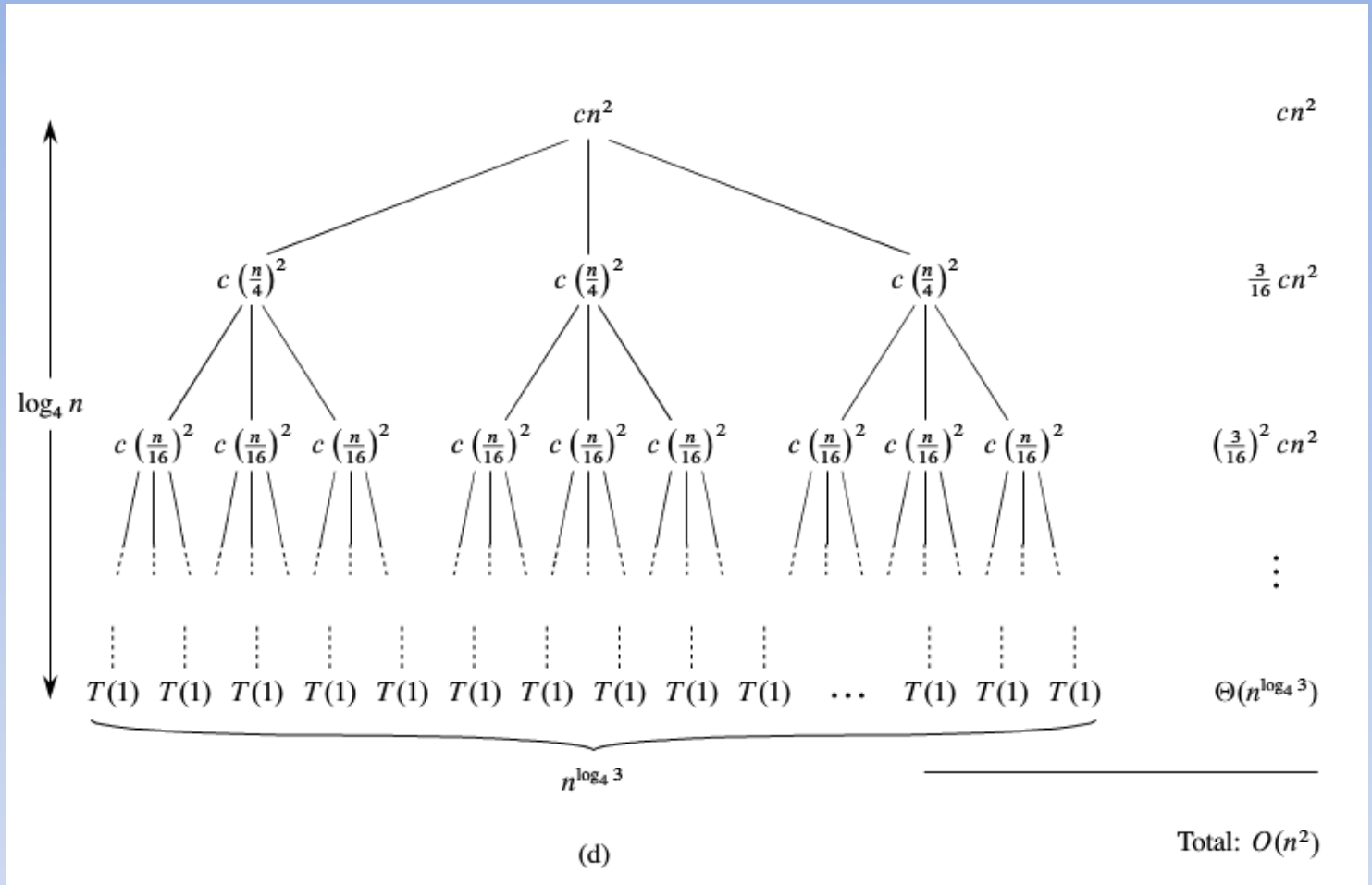
$T(N) = 3T(N/4) + n^2$


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$

(c)


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$

(d)



(d)

NOTE: Leaves = $branching^{height} = 3^{\log_4 n}$

$$3^{\log_4 n} = n^{\log_4 3}$$

$$\log_4 3^{\log_4 n} = \log_4 n^{\log_4 3}$$

$$\log_4 n \log_4 3 \quad = \log_4 3 \log_4 n$$

Uneven Tree:

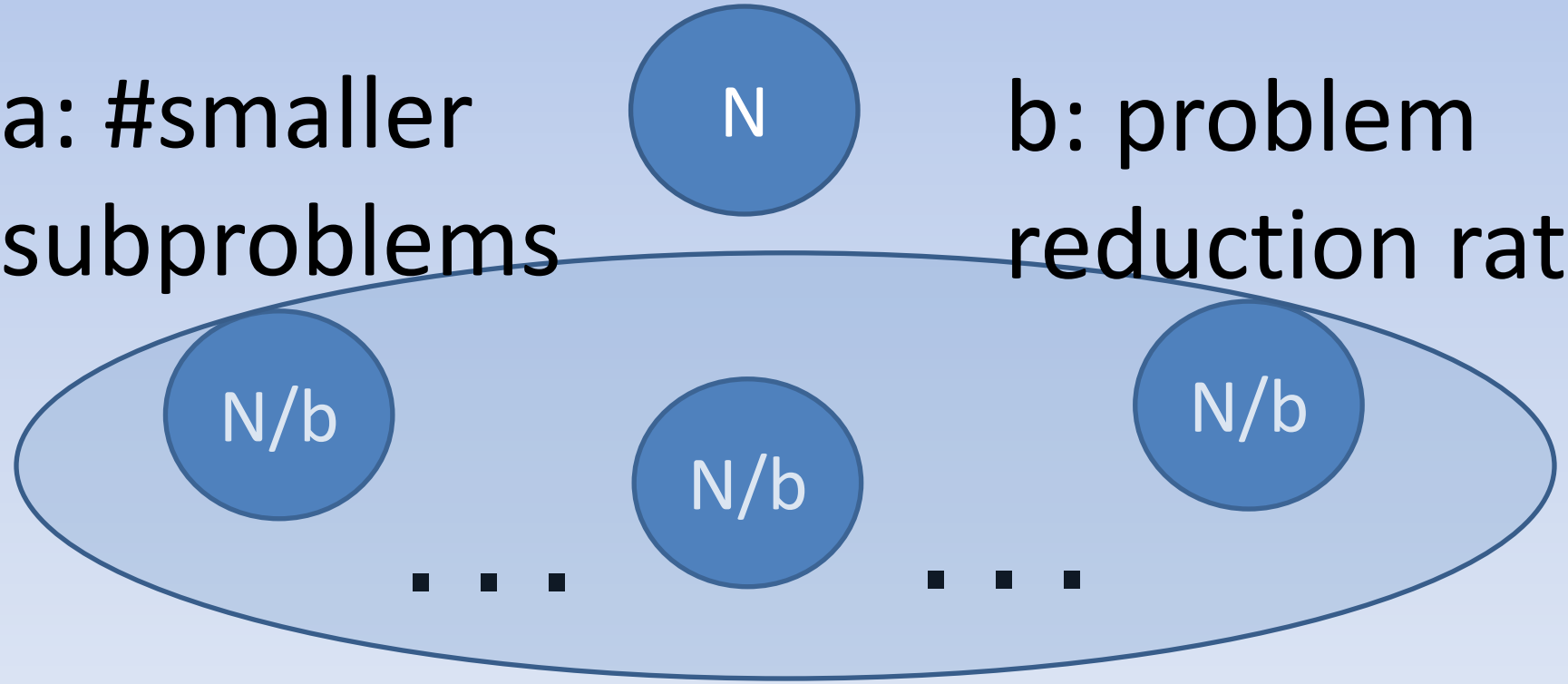
$$T(N) = T(N/3) + T(2n/3) + cn$$

Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:

a: #smaller subproblems

b: problem reduction rate



Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:
- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subproblems!
  - Book based around $n^{\log_b a}$
  - Look at relationship between $\log_b a$ and d

Solving Recurrences w/ Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

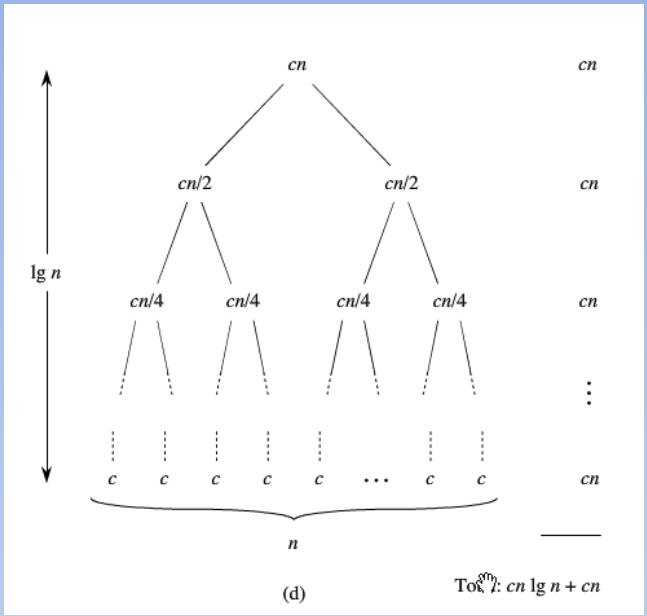
then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Solving Recurrences w/ Master Method

- Perhaps we can come up with some general formula for recurrences:
- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subproblems!
- 3 Cases to think about!

Master Method Case 1

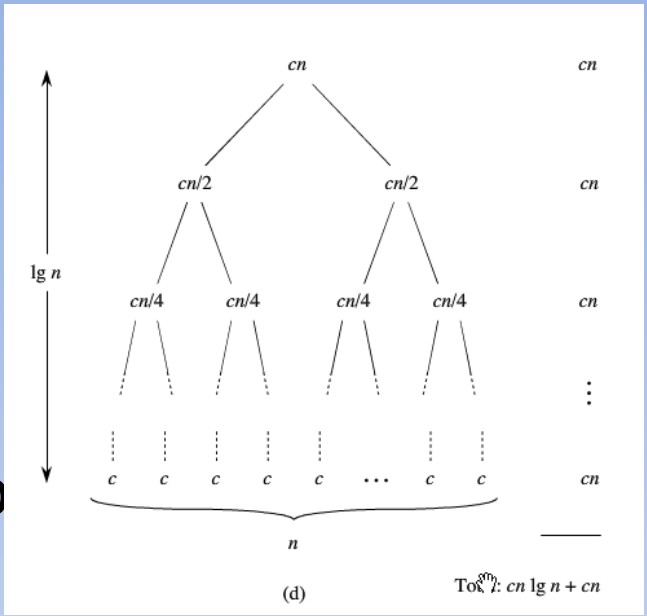


- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subproblems!

- Assume $a = b^d$
  - a (or equivalently b^d) subproblems each level
  - Each subproblem has size $\frac{N}{b^{depth}}$ and there are a^{depth}
    - if d=1, then a=b!!!
  - n^d work to merge
    - $(\frac{N}{b^{depth}})^d$
- $O(N^d \log N)$
  - Balance between work at each level and depth

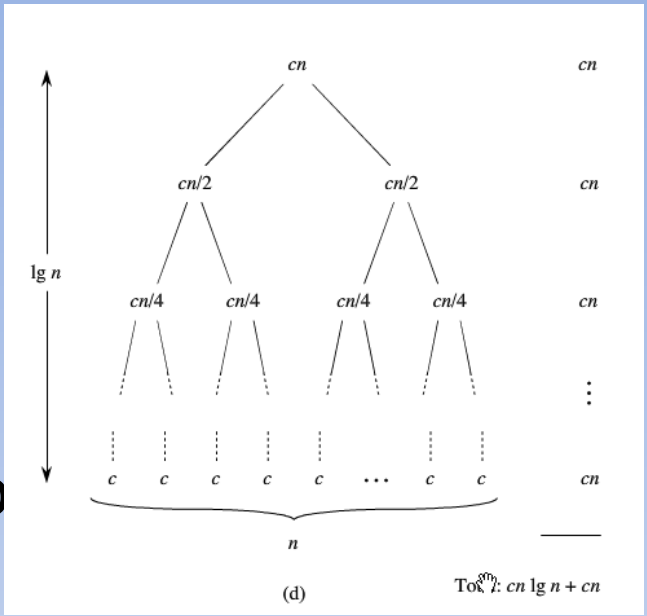
- Example:
  - a = 2, b=2 , d=1
  - $b^d = 2^1 = 2 = a$
  - $N\lg_2 N$

Master Method Case 1



- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subprob
- Assume $a = b^d$
  - a (or equivalently b^d) subproblems each level
  - Each subproblem has size $\frac{N}{b^{depth}}$ and there are a^{depth}
    - if d=1, then a=b!!!
  - n^d work to merge at each level
    - $(\frac{N}{b^{depth}})^d\ a^{depth} = N^d\ (\frac{a}{b^d})^{depth}$

Master Method Case 1



- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subprob
- Assume $a = b^d$
  - a (or equivalently b^d) subproblems each level
  - Each subproblem has size $\frac{N}{b^{depth}}$ and there are a^{depth}
    - if d=1, then a=b!!!
  - n^d work to merge at each level
    - $(\frac{N}{b^{depth}})^d\ a^{depth} = N^d\ (\frac{a}{b^d})^{depth}$

Solving Recurrences w/ Master Method

- Work(level$_j$) ≤ $a^j * c(N/b^j)^d$
  - a^j. This is the number of nodes at level j
  - $c(N/b^j)^d$: Each of the nodes do this much work
    - (N^d) where N is reduced to N/b^j
- Levels = Depth = $\log_b N$

$$Total\ Work = cN^d \cdot \sum_{j=0}^{j=\log_b N} \left(\frac{a}{b^d}\right)^j$$

- NOTE: $a = b^d$ means $a/b^d = 1$
  - Master Method case 1: $cN^d \cdot \log_b N = O(N^d \log N)$

Master Method w/ Case 2

- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subproblems!

- Assume $a < b^d$
  - Number of subproblems is growing slowly
- $O(N^d)$
  - Total work dominated by the combine step at root!
  - Good (subproblem simplification) is beating evil (subproblem proliferation)!

- $T(N) = 2T(N/2) + O(n^2)$
  - Less work is being done at each level
  - Work at level 2 is $2T(N/2) = 2(N/2)^2 = 2(N^2/4) =$ **$N^2/2$**
    - If 4 problem were being created: $4 = b^d$ so $4(N^2/4) =$ **N^2 at each level!**

- $a = 2, b=2 , d=2$
  - $a = 2 < b^d = 2^2 = 4$
- Work in total is dominated by root!
  - $O(N^2)$

Master Method w/ Case 3

- $T(N) = aT(N/b) + O(N^d)$
  - N^d work down to merge subproblems!

- Assume $a > b^d$
- $T(N) = 4T(N/2) + O(N)$
  - More work is being done at each level
    - Level 2 is now $4(N/2) =$ **2N**
      - Each level now has more work!
- $a = 4$ $b=2$, $d=1$
  - $a = 4 > b^d = 2^1 = 2$
- Work down at the leaves is overwhelming all other levels.
- How many leaves?
  - Next Slide

Master Method w/ Case 3

- Assume $a > b^d$
- $T(N) = 4T(N/2) + O(n)$
  - More work is being done at each level
- $a = 4$ $b=2$, $d=1$
  - $a = 4 > b^d = 2^1 = 2$
- How many leaves
  - Each node has 4 children
  - 4^{depth} number of leaves
- Depth = $\log_b N$
  - $4^{\log_2 N} = N^{\log_2 4}$

Master Method w/ Case 3

- Depth = $\log_b N$
  - $4^{\log_2 N} = N^{\log_2 4}$

$$4^{\log_2 N} = N\log^2 4$$

$$\log_2(4^{\log_2 N}) = \log_2(N^{\log_2 4})$$

$$\log_2 N \log_2 4 = \log_2 4 \log_2 N$$

- $4^{\log_2 N} = N^{\log_2 4} = N^2$

Solving Recurrences w/ Master Method Case 2

$$Total\ Work = cN^d \cdot \sum_{j=0}^{j=\log_b N} \left(\frac{a}{b^d}\right)^j$$

- IF: $a < b^d$ means $a/b^d < 1$
- SO: $(a/b^d)^j$ is getting smaller
- SO Asymptotically : Total work dominated by root where j=0 and $T(N) = cN^d$

Solving Recurrences w/ Master Method Case 3

$$Total\ Work = cN^d \cdot \sum_{j=0}^{j=\log_b N} \left(\frac{a}{b^d}\right)^j$$

- Finally: $a > b^d$ means $a/b^d > 1$
- SO: $(a/b^d)^j$ is growing exponentially!
- Total work is going to be dominated asymptotically by leaves:

$$Total\ Work \leq\ cN^d \cdot \left(\frac{a}{b^d}\right)^{log_b N}$$

Solving Recurrences w/ Master Method Case 3

$$Total\ Work \leq cN^d \cdot \left(\frac{a}{b^d}\right)^{log_b N}$$

$$cN^d \cdot \left(\frac{a}{b^d}\right)^{log_b N} = cN^d \cdot \mathrm{a}^{\log_\mathrm{b} \mathrm{N}} \cdot b^{-\mathrm{d}\log_\mathrm{b} \mathrm{N}}$$

$$b^{-\mathrm{d}\log_\mathrm{b} \mathrm{N}} = N^{-\mathrm{d}}$$

$$cN^d \cdot \mathrm{a}^{\log_\mathrm{b} \mathrm{N}} \cdot b^{-\mathrm{d}\log_\mathrm{b} \mathrm{N}} = c\,\frac{N^d}{N^{\mathrm{d}}} \cdot \mathrm{a}^{\log_\mathrm{b} \mathrm{N}}$$

$$O(\mathrm{a}^{\log_\mathrm{b} \mathrm{N}})$$

Solving Recurrences w/ Master Method Case 3

$$Total\ Work =$$

$$O\left(\mathrm{a}^{\log_{\mathrm{b}}\mathrm{n}}\right) = O\left(n^{\log_{\mathrm{b}}\mathrm{a}}\right)$$

$$log_b(\mathrm{a}^{\log_{\mathrm{b}}\mathrm{n}}) = log_b(n^{\log_{\mathrm{b}}\mathrm{a}})$$

$$log_b n \cdot log_b \mathrm{a} \quad = log_b a \cdot log_b n$$

Total Work = $O\left(n^{\log_{\mathrm{b}}\mathrm{a}}\right)$

Textbook

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

- Consider $f(n) = n^d$
- Case 1: $n^d < cn^{\lg_b a}$
  - $\log_n(n^d) < \log_n(n^{\lg_b a})$
  - $d < \lg_b a$
  - $b^d < b^{\lg_b a}$
  - $b^d < a$
- $\Theta(n^{\log_b a})$
  - Case 3 from earlier!
  - Work dominated by leaves!!

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

- Consider $f(n) = n^d$
- Case 3: $f(n) = \Omega(n^{\log_b a + \in})$
- $n^d > cn^{\log_b a + \in}$
  - $\log_n(n^d) > \log_n(n^{\log_b a + \in}) \geq \log_n(n^{\log_b a})$
  - $d > \lg_b a$
  - $b^d > b^{\lg_b a}$
  - $b^d > a$
- $\Theta(f(n)) = O(n^d)$

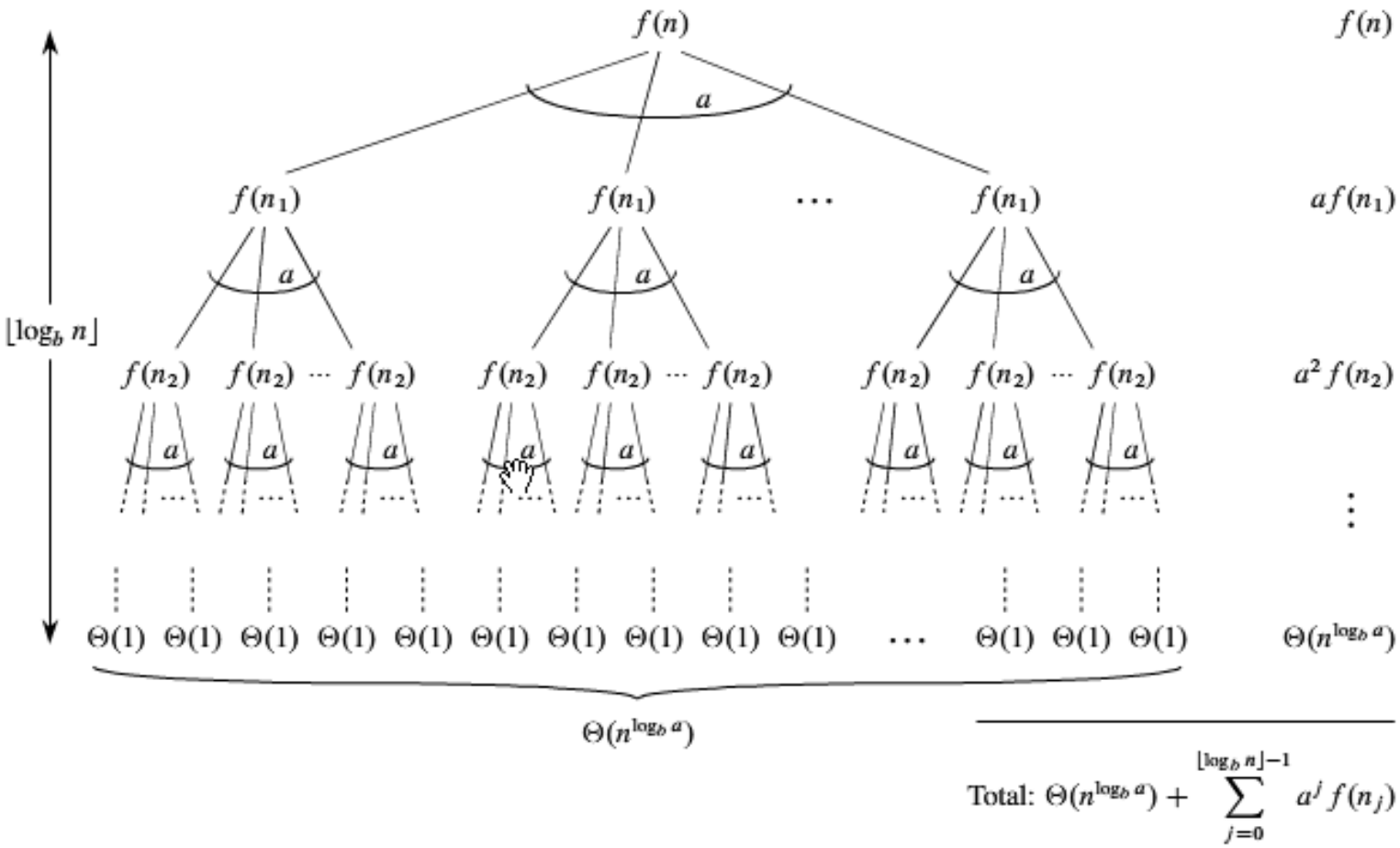


Figure 4.8 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.27).

Solving Recurrences w/ Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

(5) Use the Master Method to solve the recurrences below?

(a) $T(N) = 3*T(N/3) + 7$

(b) $T(N) = 4*T(N/4) + N$

(c) $T(N) = 3*T(N/4) + 7$

Solving Recurrences w/ Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

$T(N) = a\,T(N/b) + O(N^d)$

$a = b^d \mid N^d * \lg(N)$

$a > b^d \mid O(n^{\log_b a}) = \mid O(a^{\log_b N})$

$a < b^d \mid O(n^d)$

(a) $T(N) = 3*T(N/3) + 7$

a= 3, b=3, d=0 | (a=3) > (b^d=1)

n^log_3(3) = O(N)

- T(N) = 4*T(N/4) + N
- **a= 4, b=4, d=1 | (a=4) = (b^d=4)**
- **NlgN**

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

- T(N) = 3*T(N/4) + 7
- **a= 3, b=4, d=0 | (a=3) > (b^d=1)**
- **N^log_4(3) = O($N^{log_4(3)}$)**

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

(d) $T(N) = 4*T(N/3) + N^2$

(e) $T(N) = 4*T(N/3) + 5$

- $T(N) = 4*T(N/3) + N^2$
- **a= 4, b=3, d=2 | (a=4) < (b^d=9)**
- **$O(N^2)$**

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

- T(N) = 4*T(N/3) + 5
- **a= 4, b=3, d=0 | (a=4) > (b^d=1)**
- **N^log_3(4) = O(N$^{\log_3(4)}$)**

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Midterm Results

(1) Use the Master Method to solve the recurrences below?

(a) $T(N) = 3*T(N/2) + N^3$

(b) $T(N) = 5*T(5N/8) + 8$

(c) $T(N) = 64*T(N/4) + N^3$

- (a) $T(N) = 3T(N/2) + N^3$
  - a=3
  - b=2
  - d=3
  - a=3 < 8=b^d

$$T(N) = O(N^d)$$
$$T(N) = O(N^3)$$

- (b) T(N) = 5T(5N/8) + 8
  - a=5
  - b=(8/5)
  - d=0
  - a=5 > 1=b^d

$$T(N) = O(N^{\log_b a})$$
$$T(N) = O(N^{\log_{8/5} 5})$$

- (c) $T(N) = 64T(N/4) + N^3$
  - a=64
  - b=4
  - d=3
  - a=64 = 64=b^d

$$T(N) = O(N^d \log N)$$
$$T(N) = O(N^3 \log N)$$

VI Graph Algorithms

Introduction *587*

22 Elementary Graph Algorithms *589*

22.1 Representations of graphs *589*
22.2 Breadth-first search *594*
22.3 Depth-first search *603*
22.4 Topological sort *612*
22.5 Strongly connected components *615*

23 Minimum Spanning Trees *624*

23.1 Growing a minimum spanning tree *625*
23.2 The algorithms of Kruskal and Prim *631*

24 Single-Source Shortest Paths *643*

24.1 The Bellman-Ford algorithm *651*
24.2 Single-source shortest paths in directed acyclic graphs *655*
24.3 Dijkstra's algorithm *658*
24.4 Difference constraints and shortest paths *664*
24.5 Proofs of shortest-paths properties *671*

25 All-Pairs Shortest Paths *684*

25.1 Shortest paths and matrix multiplication *686*
25.2 The Floyd-Warshall algorithm *693*
25.3 Johnson's algorithm for sparse graphs *700*

26 Maximum Flow *708*

26.1 Flow networks *709*
26.2 The Ford-Fulkerson method *714*
26.3 Maximum bipartite matching *732*
★ 26.4 Push-relabel algorithms *736*
★ 26.5 The relabel-to-front algorithm *748*

- Given Graph G with V={1,2,3,4,5,6,7,8} and Adjacency List below:
  - ❖1 ==> [2, 4]
  - ❖2 ==> [3]
  - ❖4 ==> [3, 8]
  - ❖5 ==> [6, 7]
  - ❖6 ==> [7]
  - ❖8 ==> [7]

BFS

- Execute BFS(G, 1) and determine the order in which the vertices are explored and the distance from the start vertex 1. Place the information into table below:

Vertex	Order Explored	Distance from 1
1		
2		
3		
4		
5		
6		
7		
8		

DFS

- Execute DFS(G) and determine the Discover time and Finish Time for each vertex and then list the vertices in topological order. Put this information into the table below:

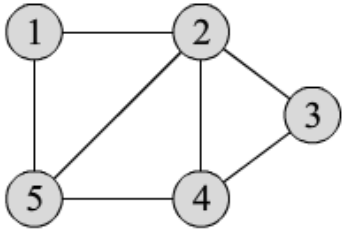
Vertex	Discover Time	Finish Time	Topological Order
1			
2			
3			
4			
5			
6			
7			
8			

Edge Types

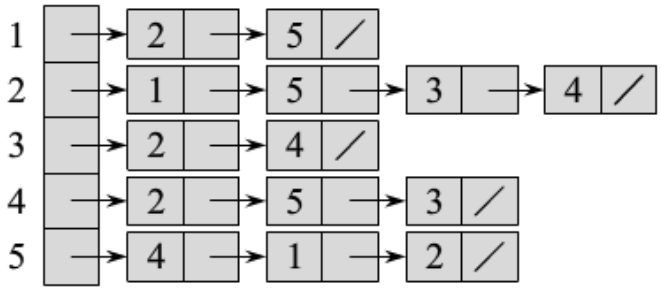
- Use the information from previous DFS question to determine the edge type for each edge of the graph. Put this information into the table below:

Edge	Edge Type
(1, 2)	
(1, 4)	
(2, 3)	
(4, 3)	
(4, 8)	
(5, 6)	
(5, 7)	
(6, 7)	
(8, 7)	

Graph Representations
Undirected



(a)

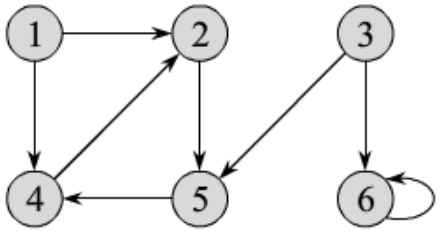


(b)

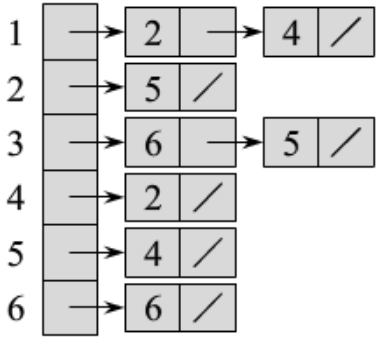
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Graph Representations
Directed



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

BFS

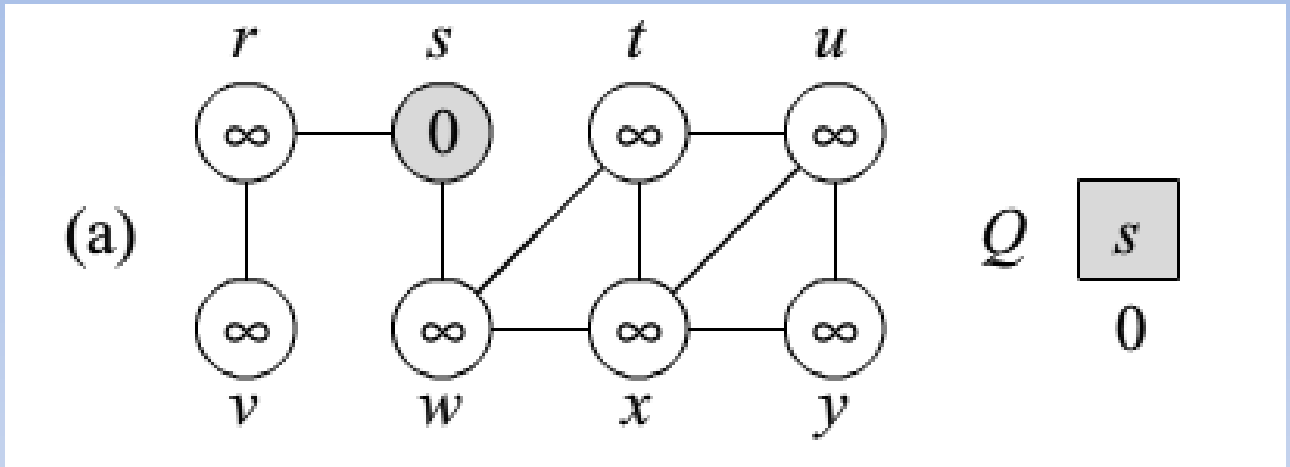
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

BFS

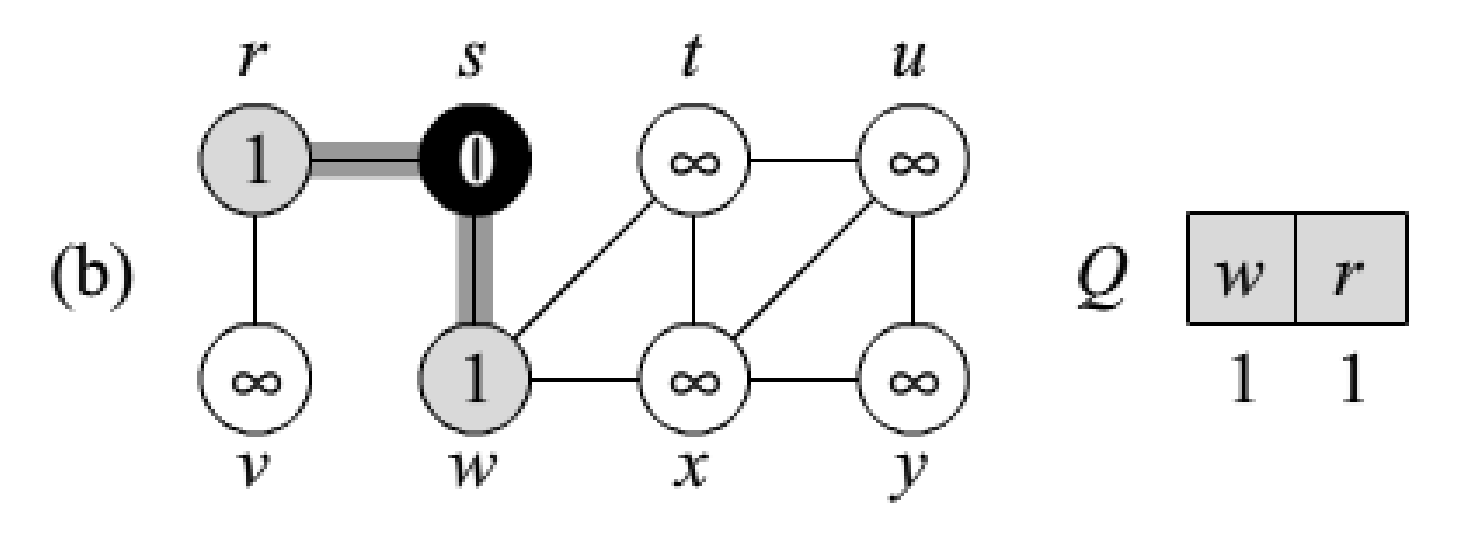
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Predecessor Subgraph

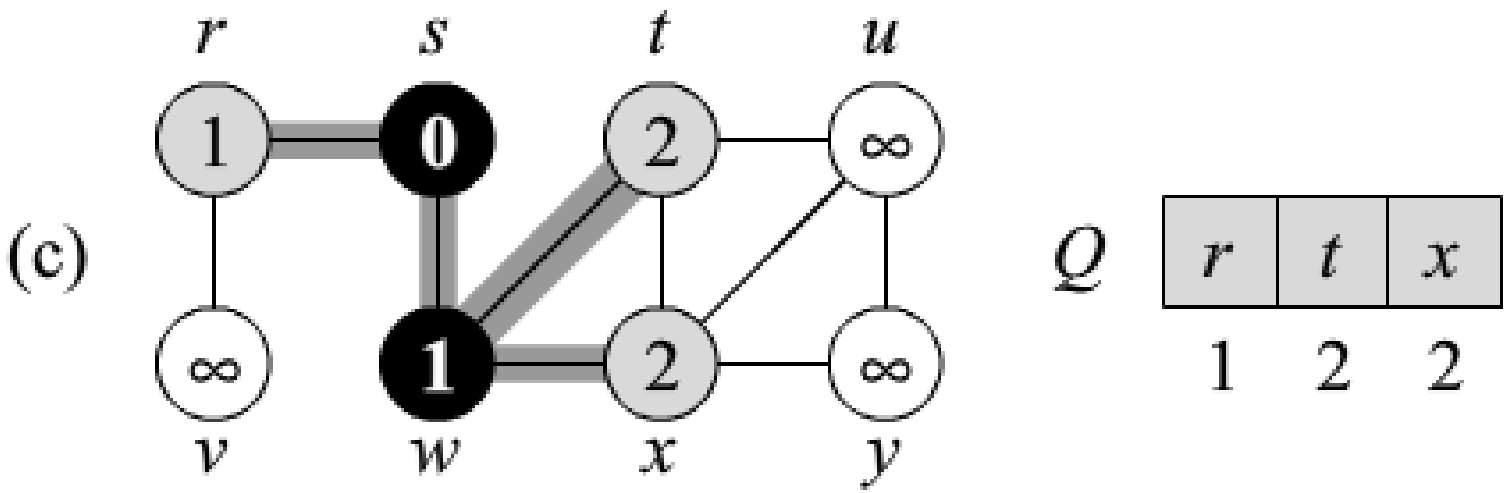
BFS
Example



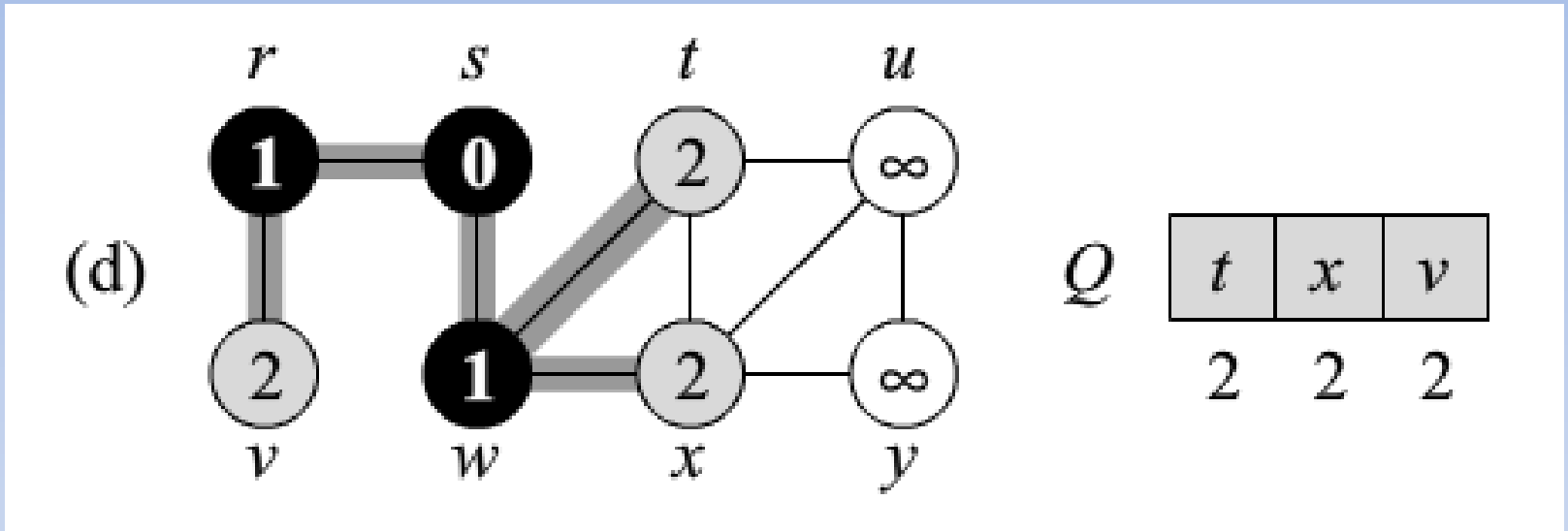
BFS
Example



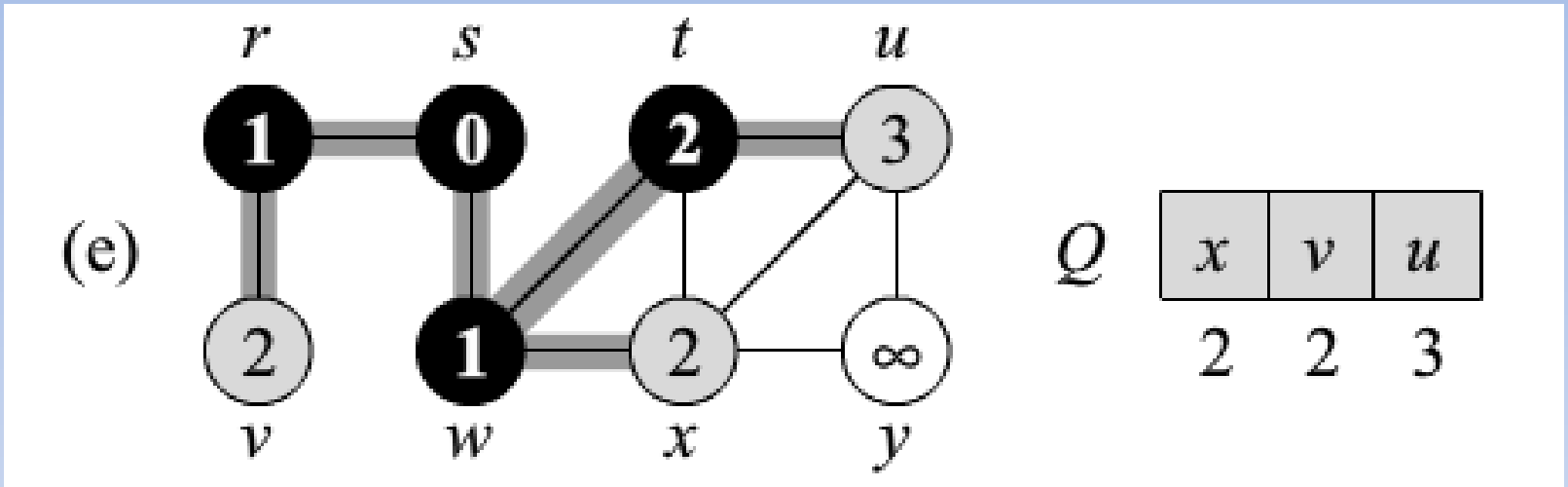
BFS
Example



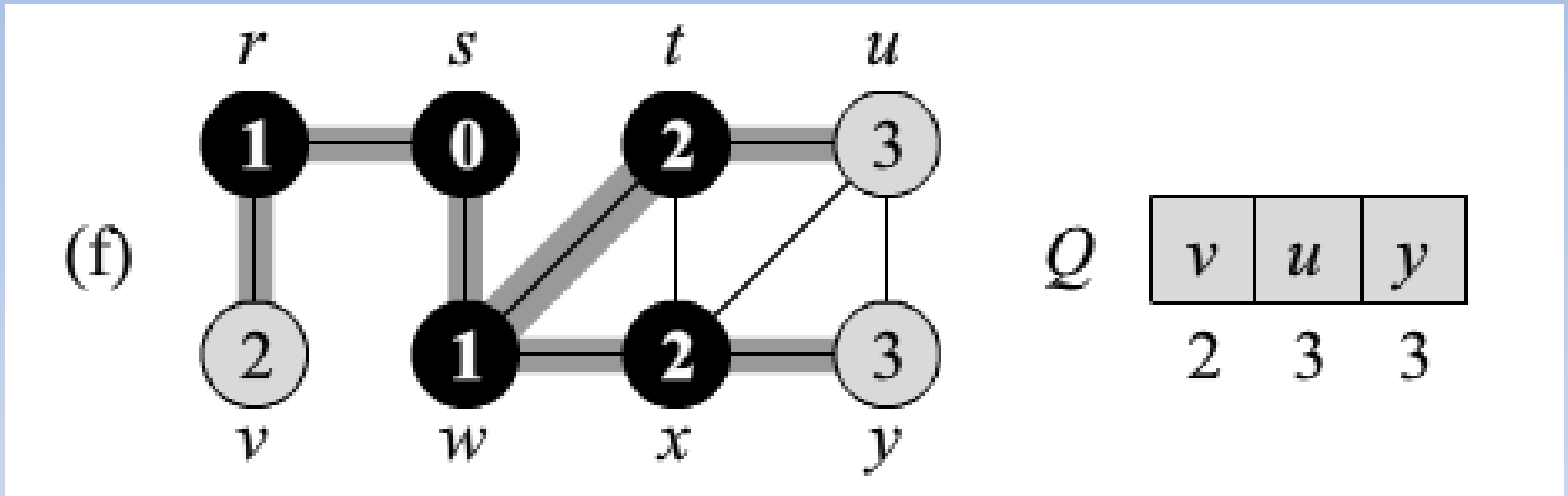
BFS
Example



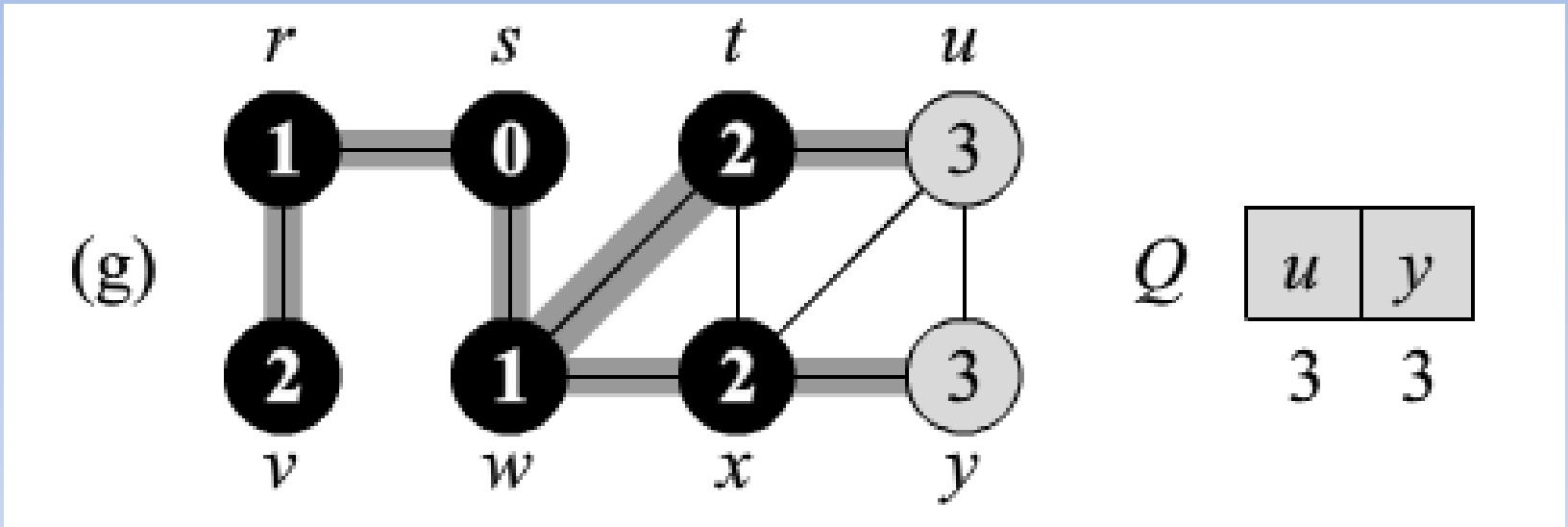
BFS
Example



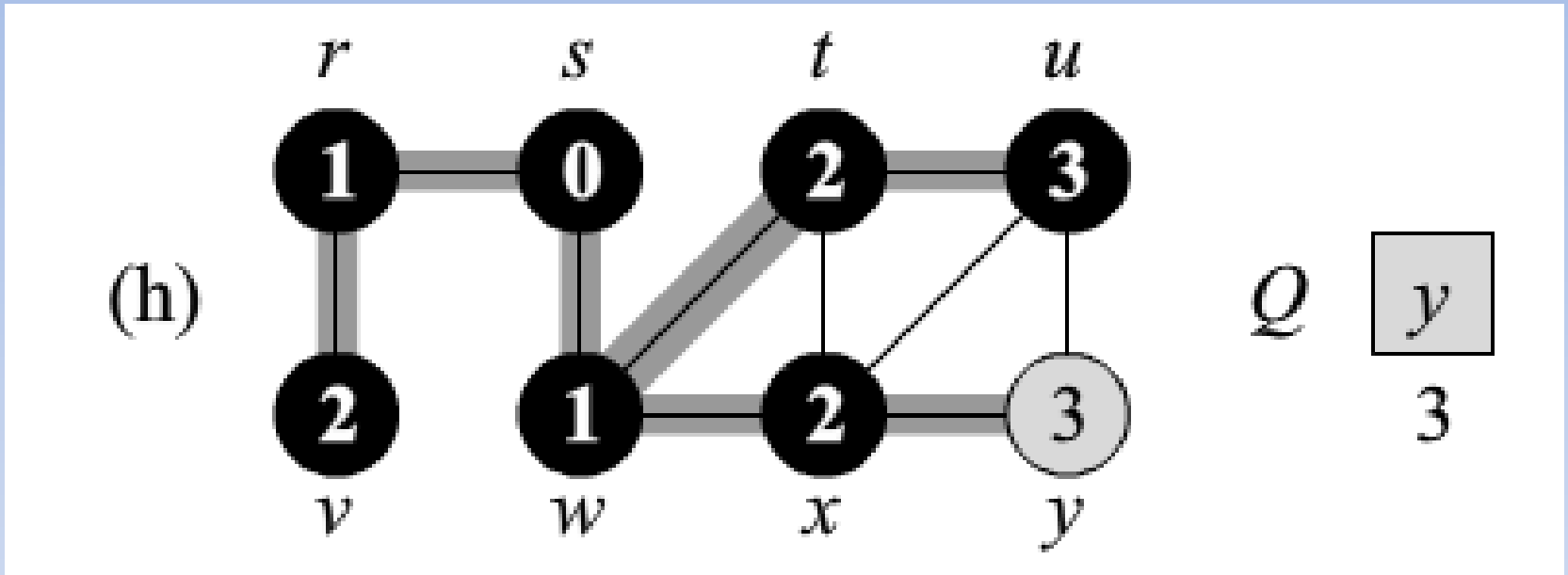
BFS
Example



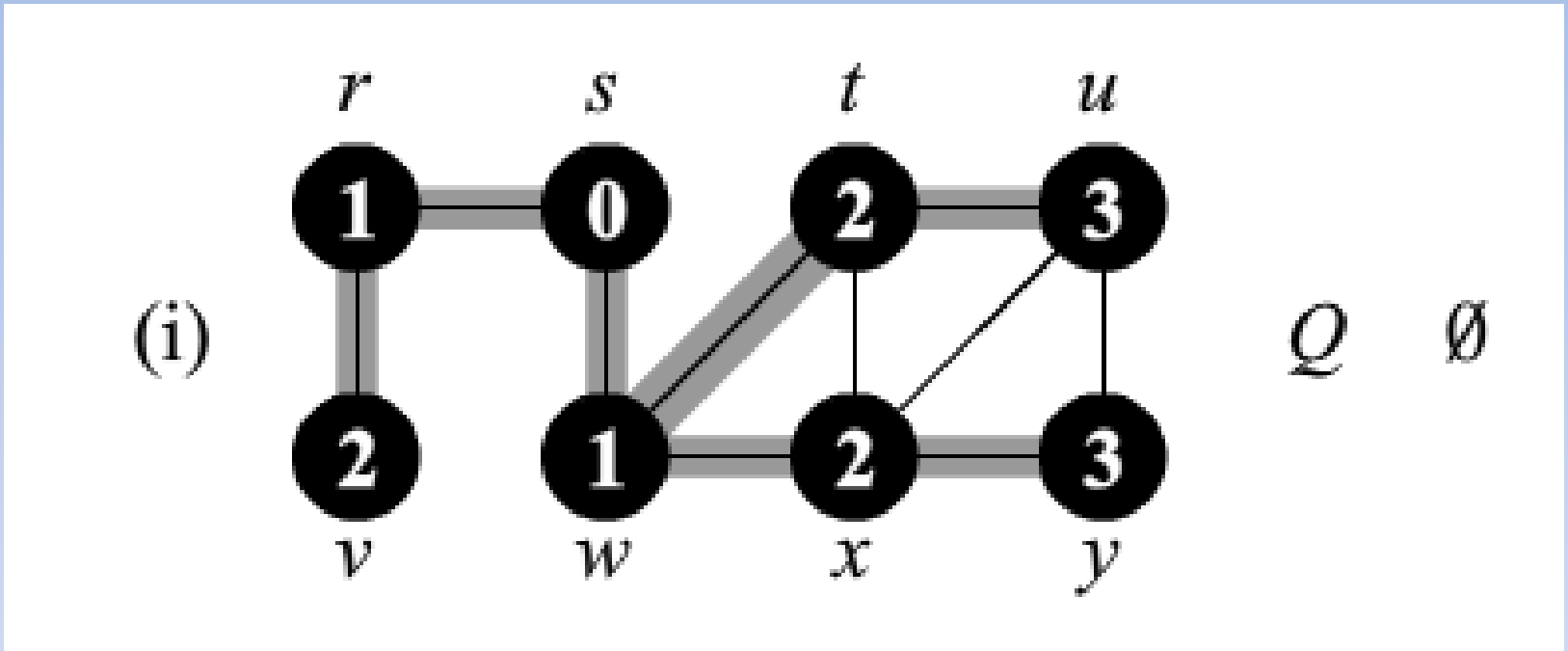
BFS
Example



BFS
Example



BFS
Example

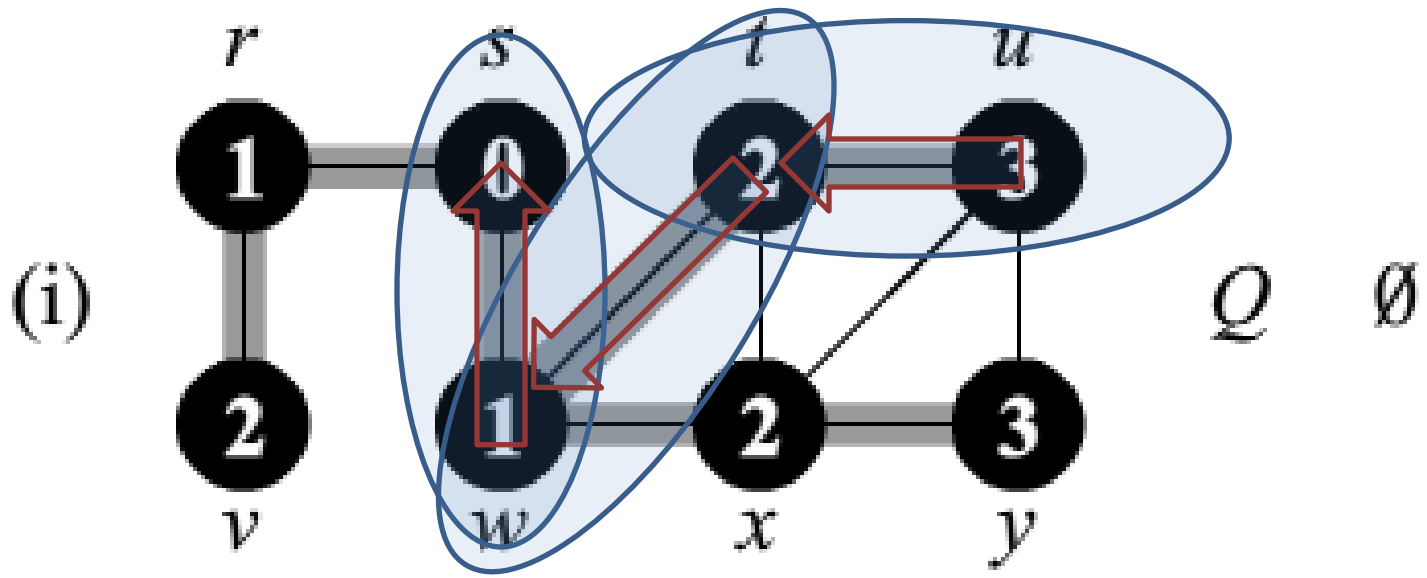


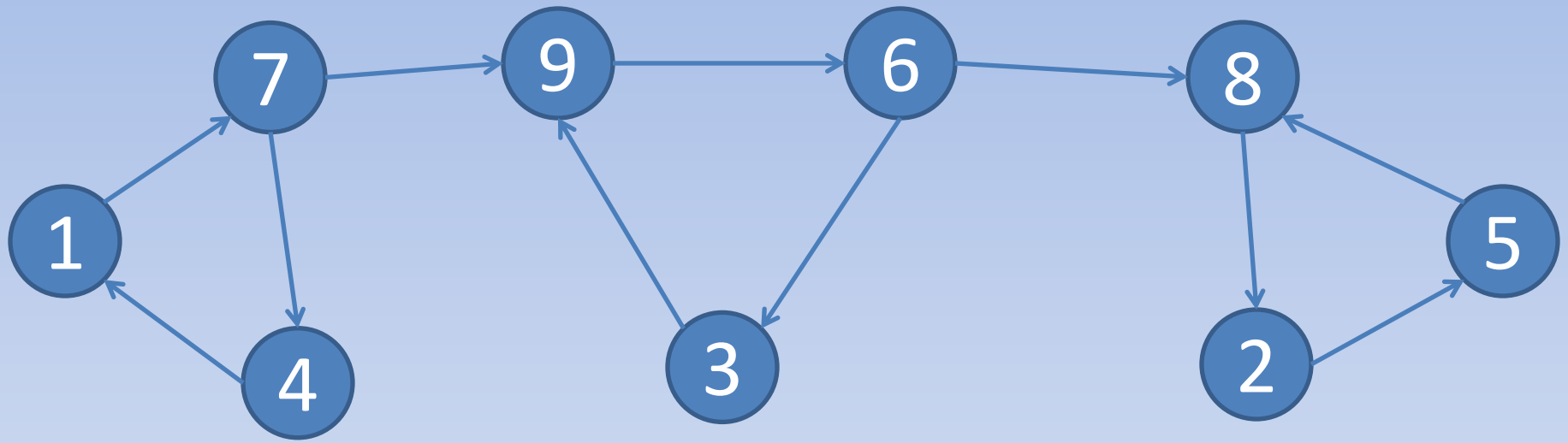
Shortest Path

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

Shortest Path: s->u

```
PRINT-PATH(G, s, v)
    if v == s
        print s
    elseif v.π == NIL
        print "no path from" s "to" v "exists"
    else PRINT-PATH(G, s, v.π)
        print v
```





12 lines (11 sloc) | 44 Bytes

Raw | Blame | History

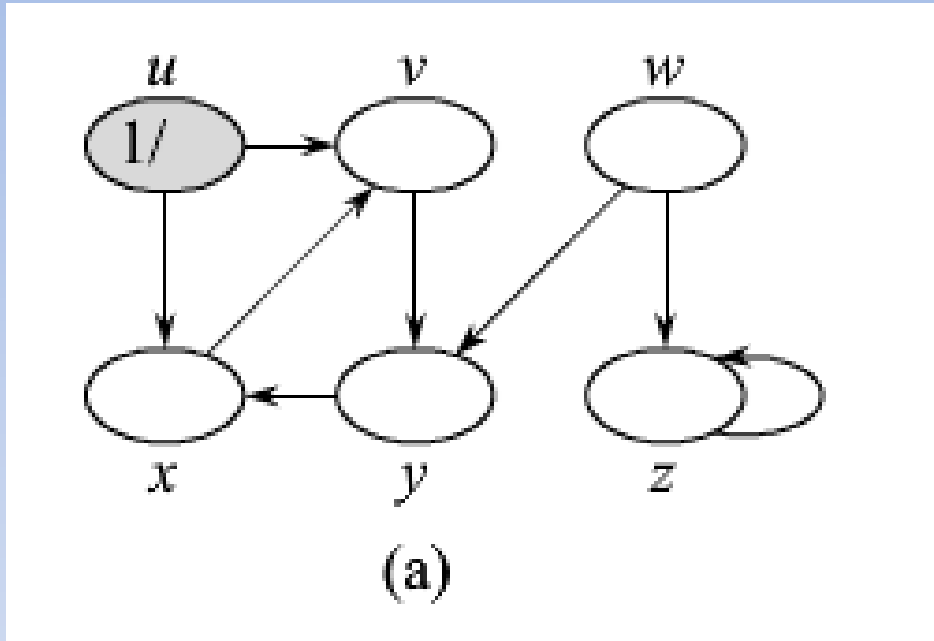
```
1 7
7 4
4 1
7 9
9 6
6 3
3 9
6 8
8 2
2 5
5 8
```

DFS

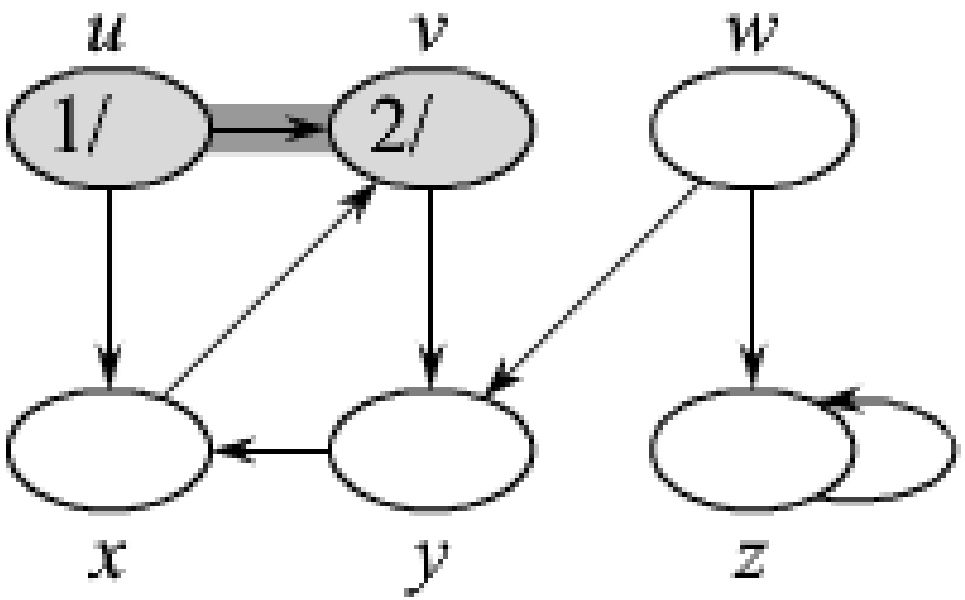
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                    // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]              // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                    // blacken u; it is finished
time = time + 1
u.f = time
```

DFS
Example

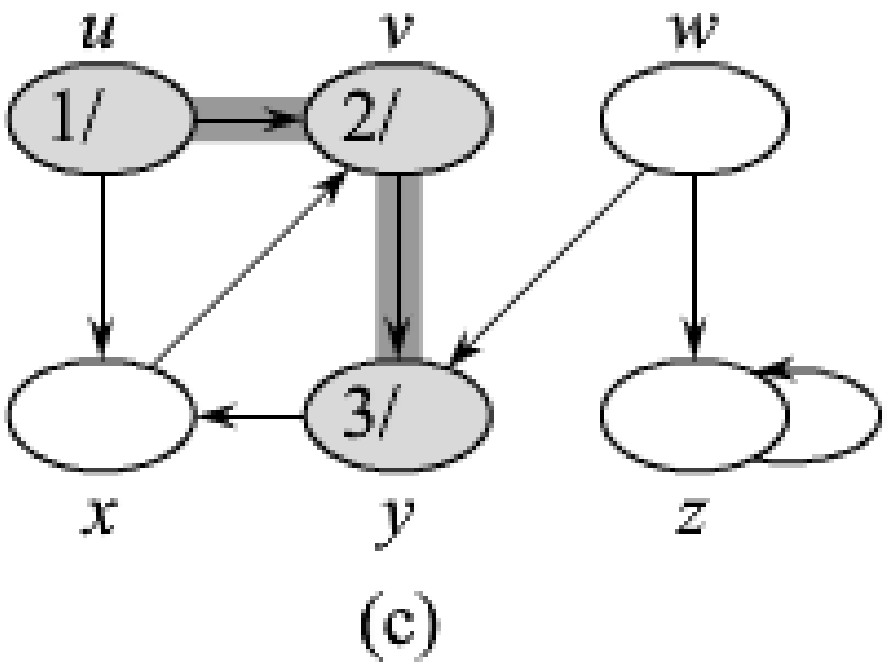


DFS
Example



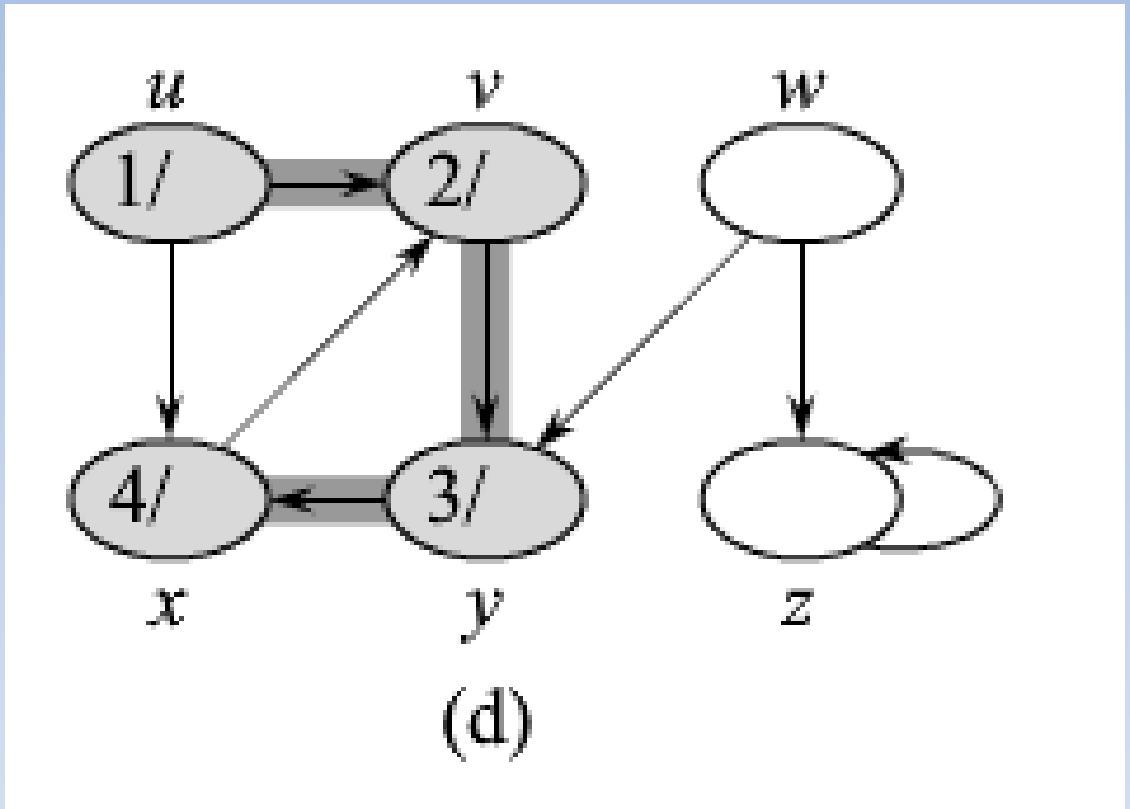
(b)

DFS
Example

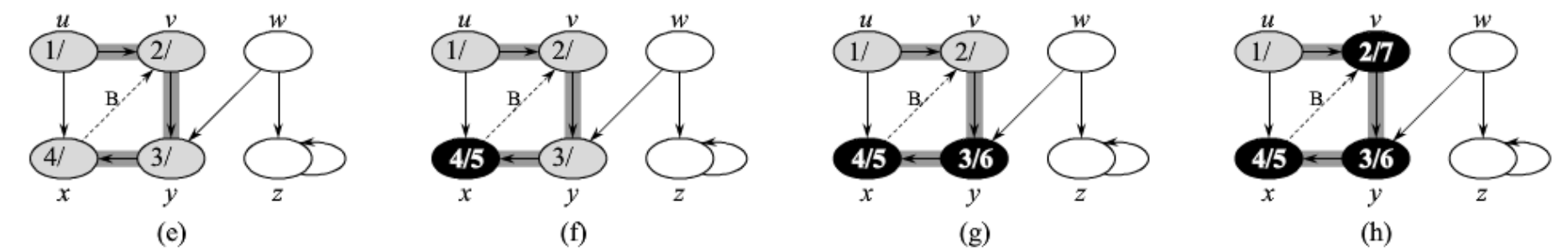


(c)

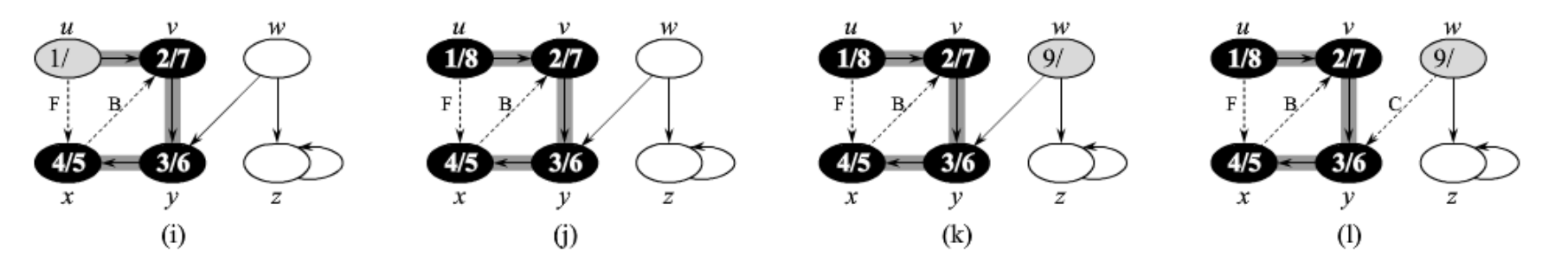
DFS
Example



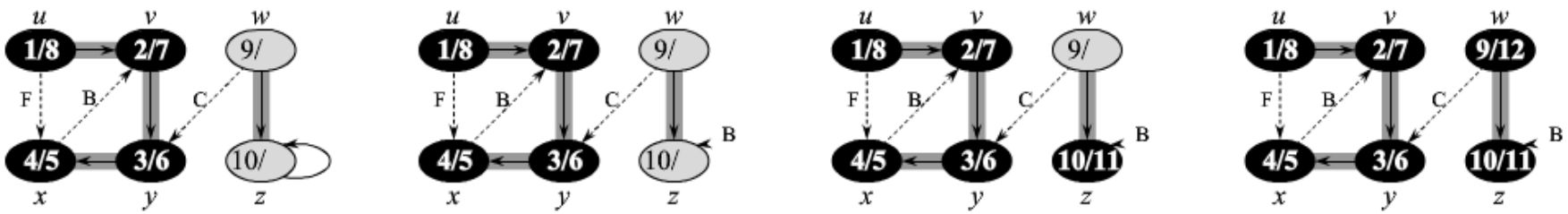
DFS
Example


u
v
w
1/
2/
B
4/
3/
x
y
z
(e)
u
v
w
1/
2/
B
4/5
3/
x
y
z
(f)
u
v
w
1/
2/
B
4/5
3/6
x
y
z
(g)
u
v
w
1/
2/7
B
4/5
3/6
x
y
z
(h)

DFS
Example


u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z
(i)
u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z
(j)
u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z
(k)
u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
x
y
z
(l)

DFS
Example


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
x
y
z
(m)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/
x
y
z
(n)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/11
x
y
z
(o)
u
v
w
1/8
2/7
9/12
F
B
C
B
4/5
3/6
10/11
x
y
z
(p)

DFS

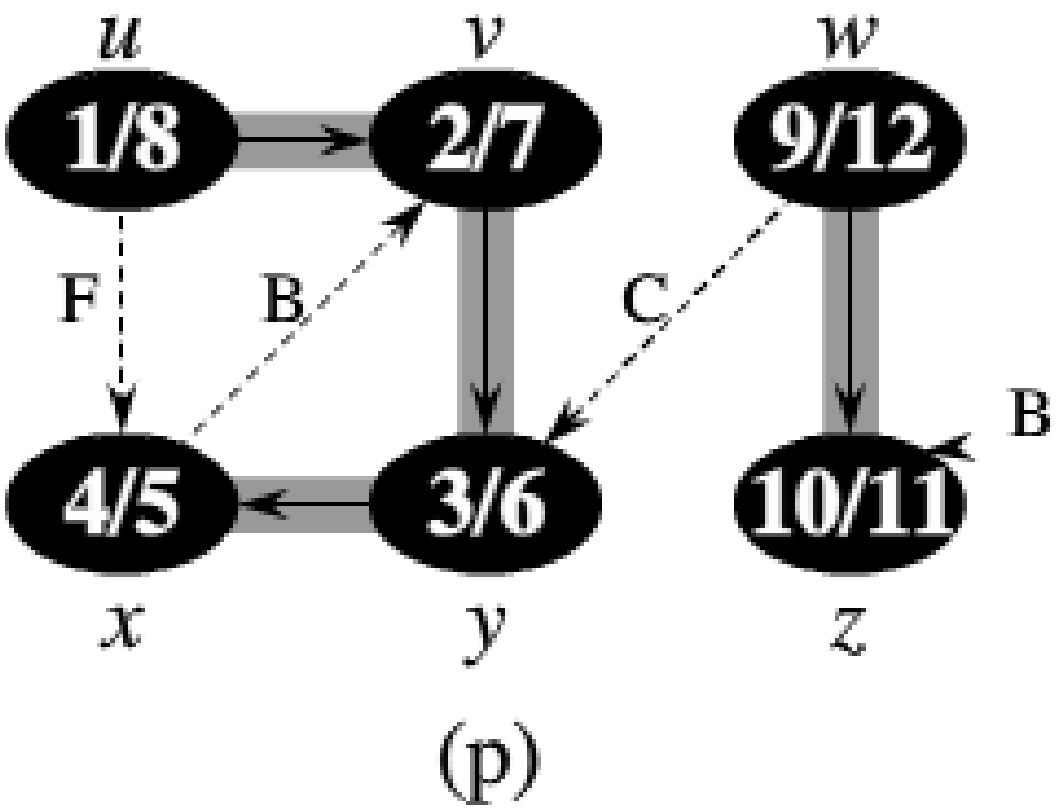
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                    // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]              // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                    // blacken u; it is finished
time = time + 1
u.f = time
```

Edge Types

- Tree Edges: Edges in the depth-first forest
- Back Edges: Nontree edges connecting a vertex to a ancestor
- Forward Edges: Nontree edges connecting a vertex to a descendant
- Cross Edges: All others (generalized cousins).

Edge Types

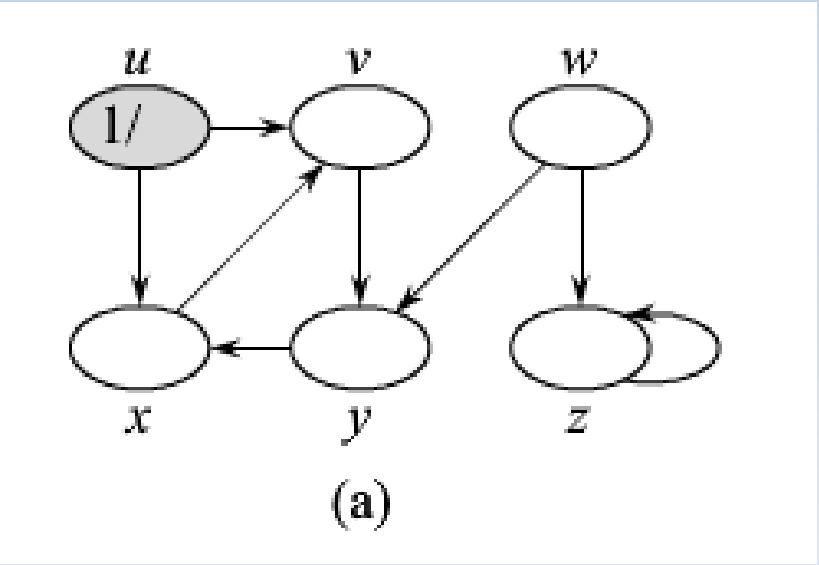


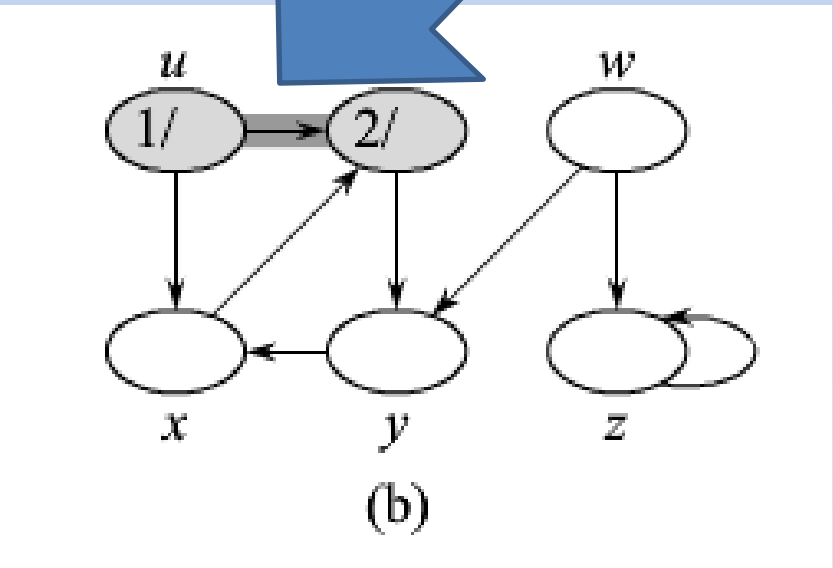
Determining Edge Types

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

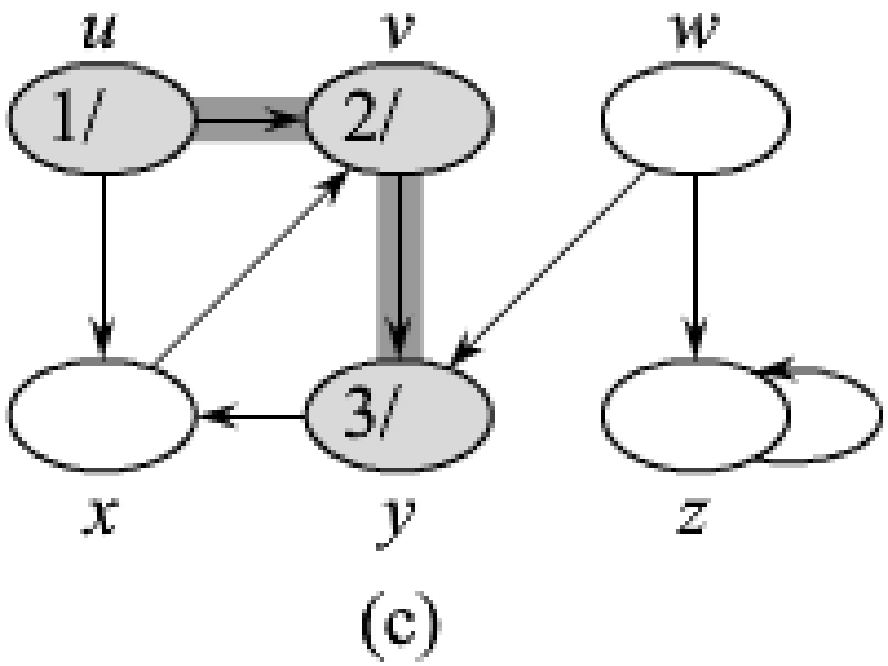
Determining Edge Types: Tree Edge

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge



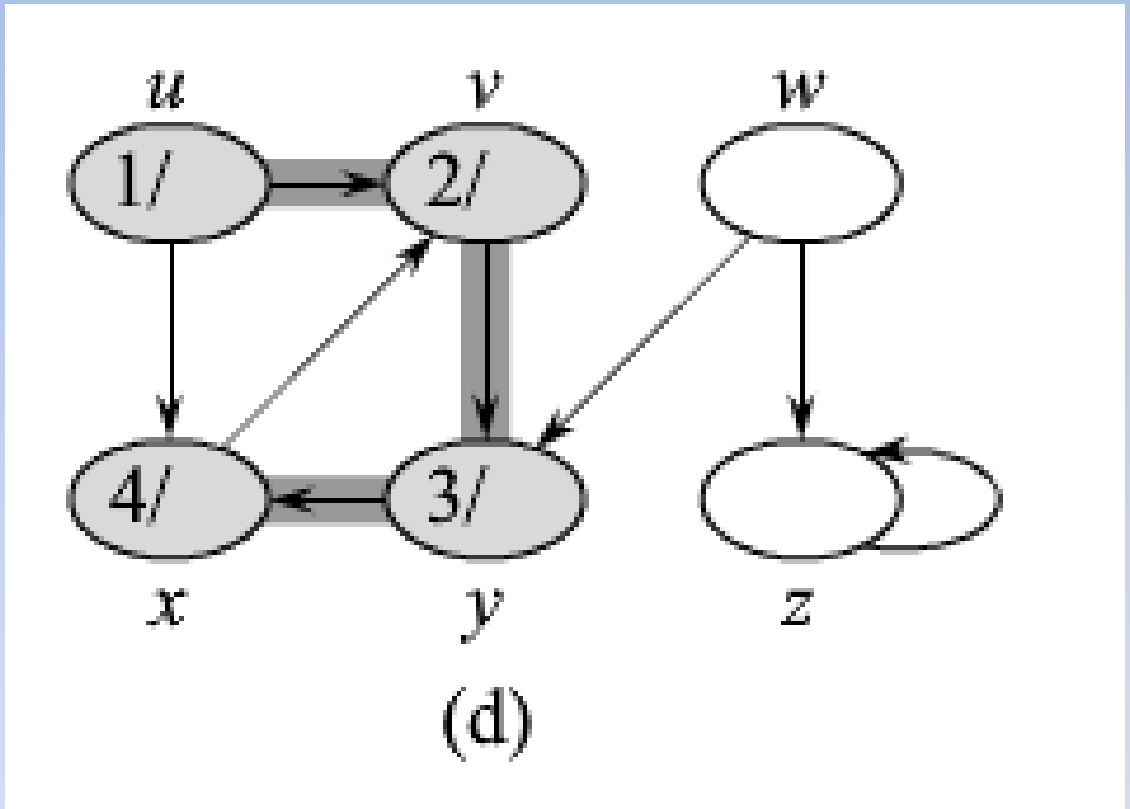


Determining Edge Types: Tree Edge



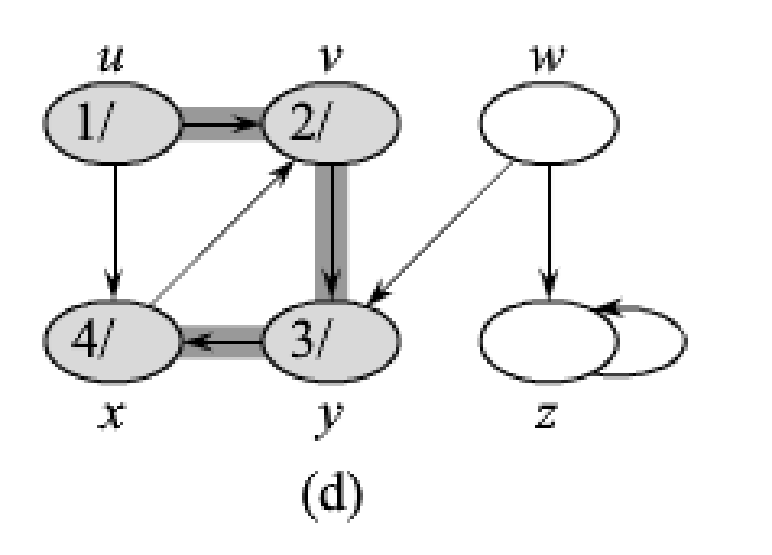
(c)

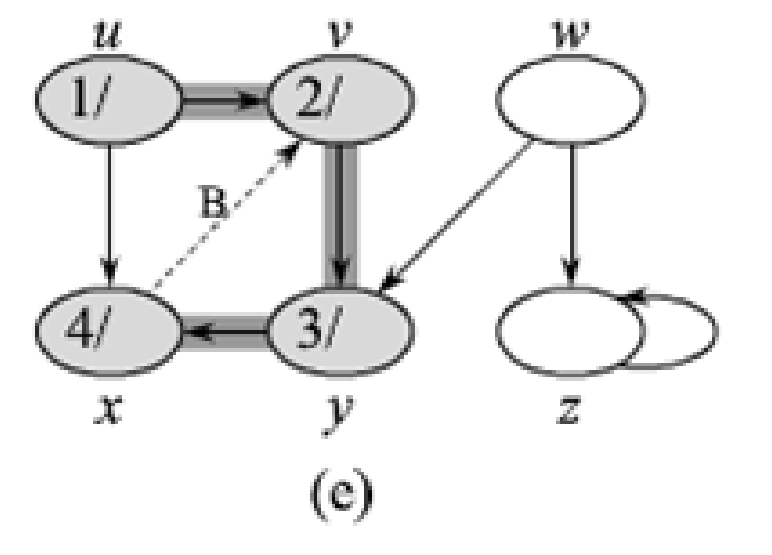
Determining Edge Types: Tree Edge



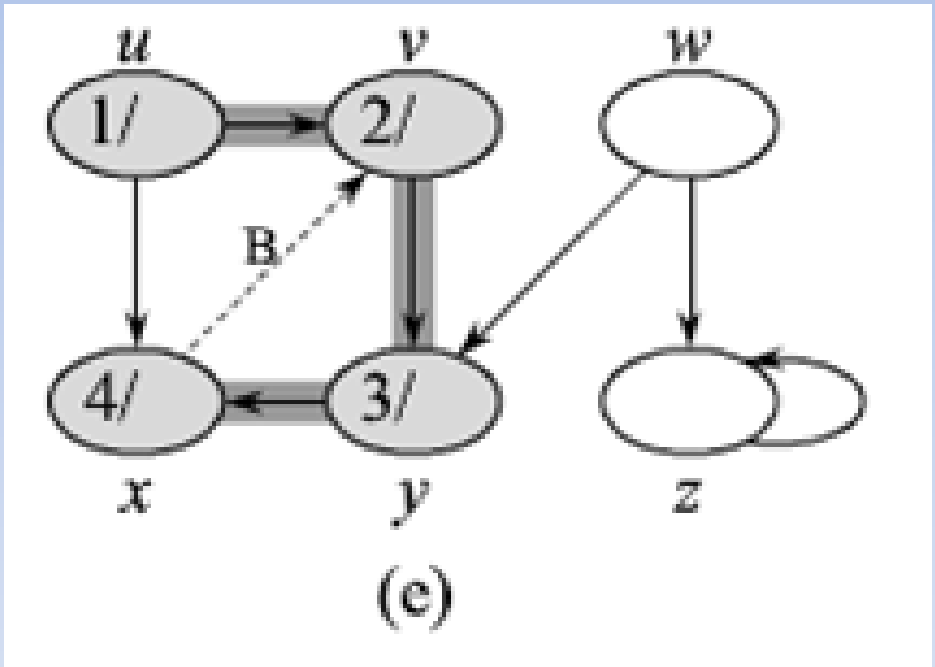
Determining Edge Types: Back Edge

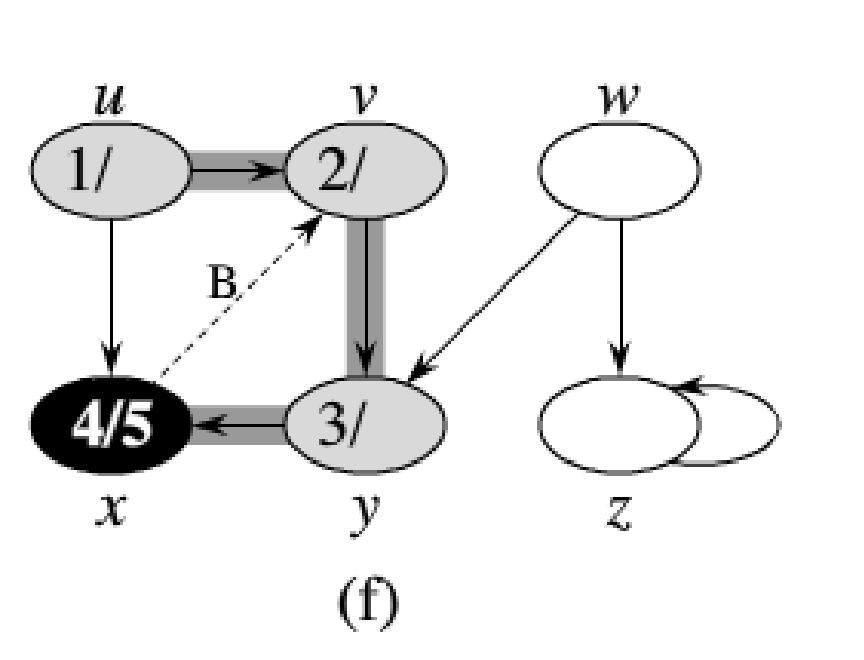
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - **GRAY: back edge**



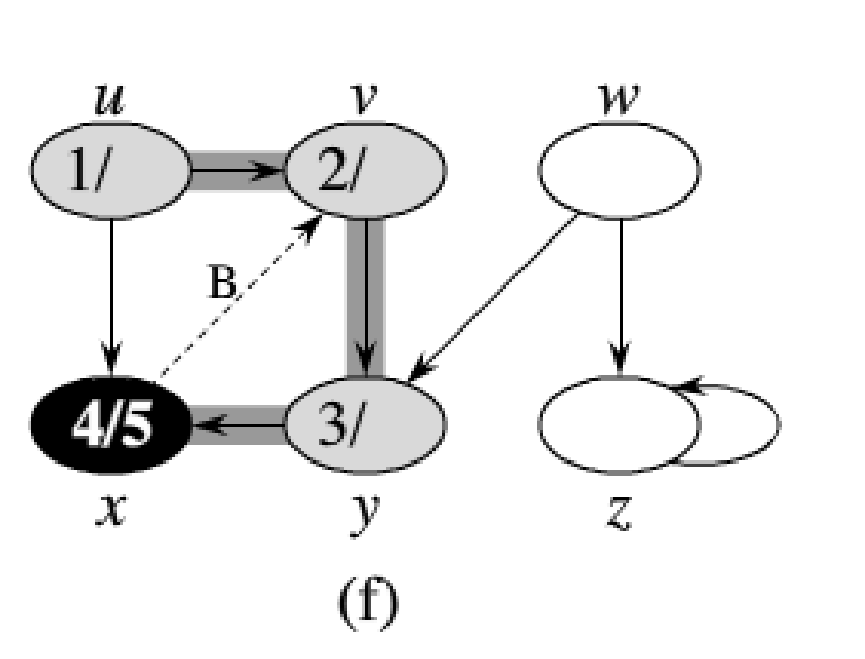


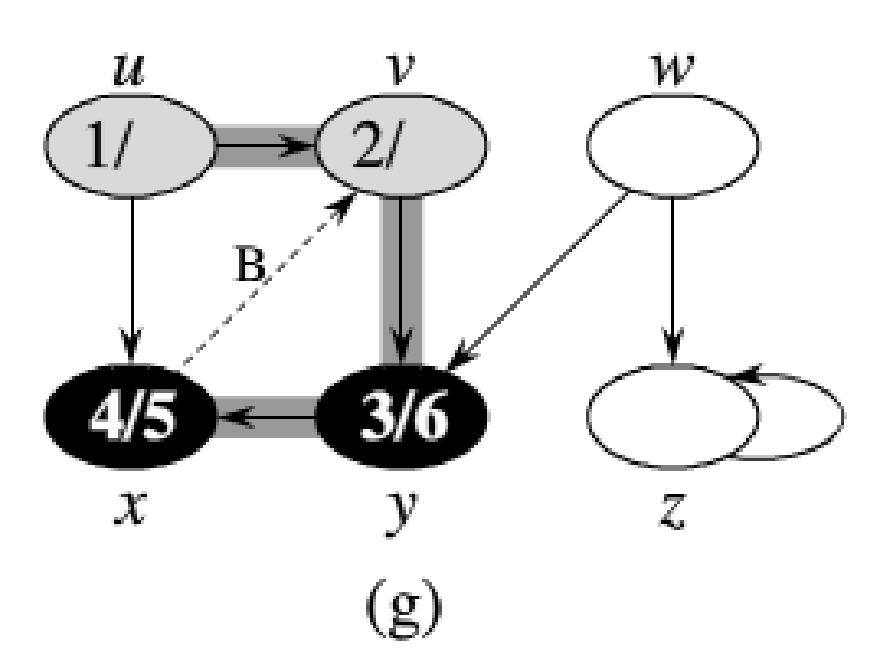
DFS Example:
X Finishes



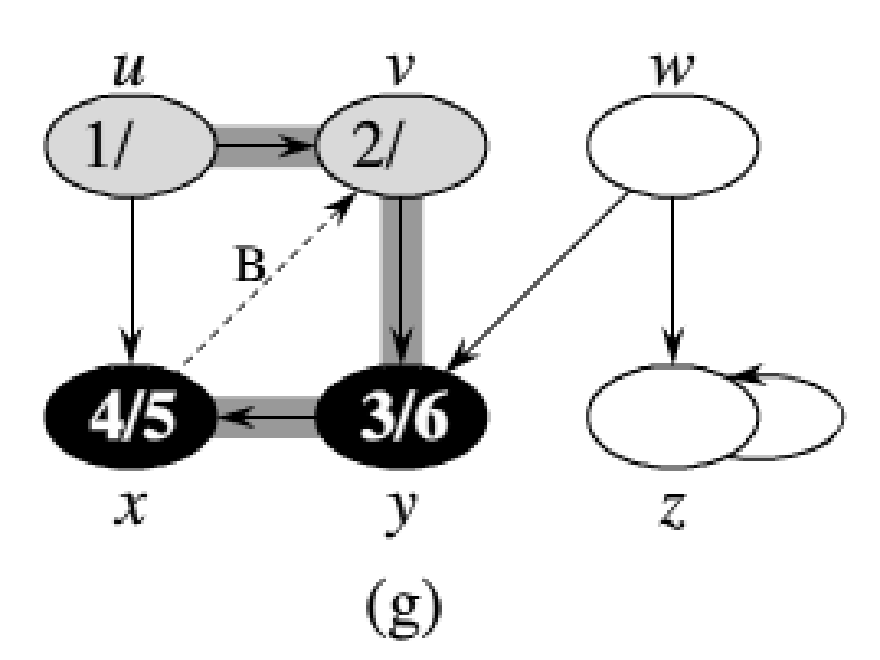


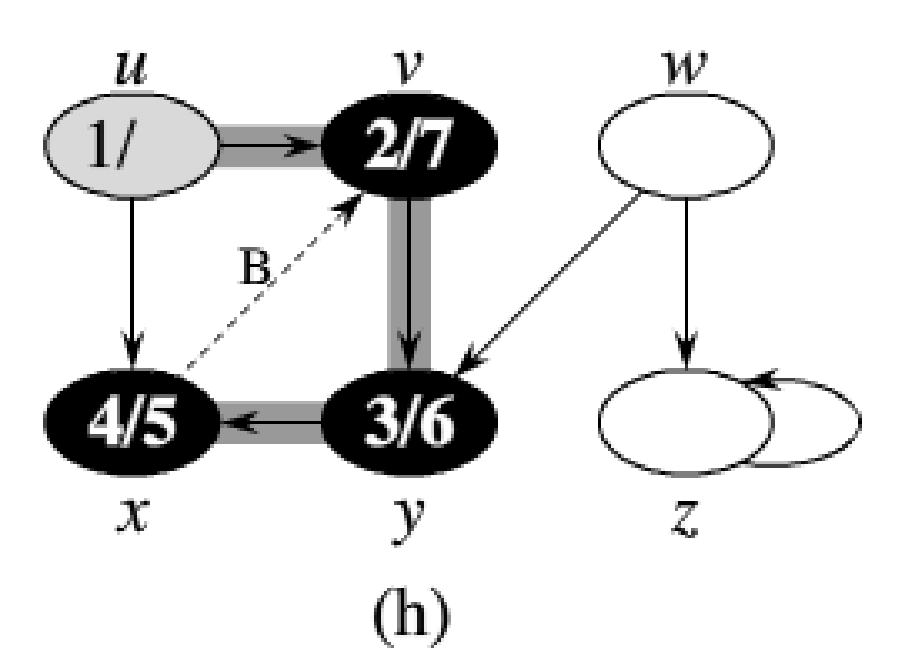
DFS Example:
y Finishes





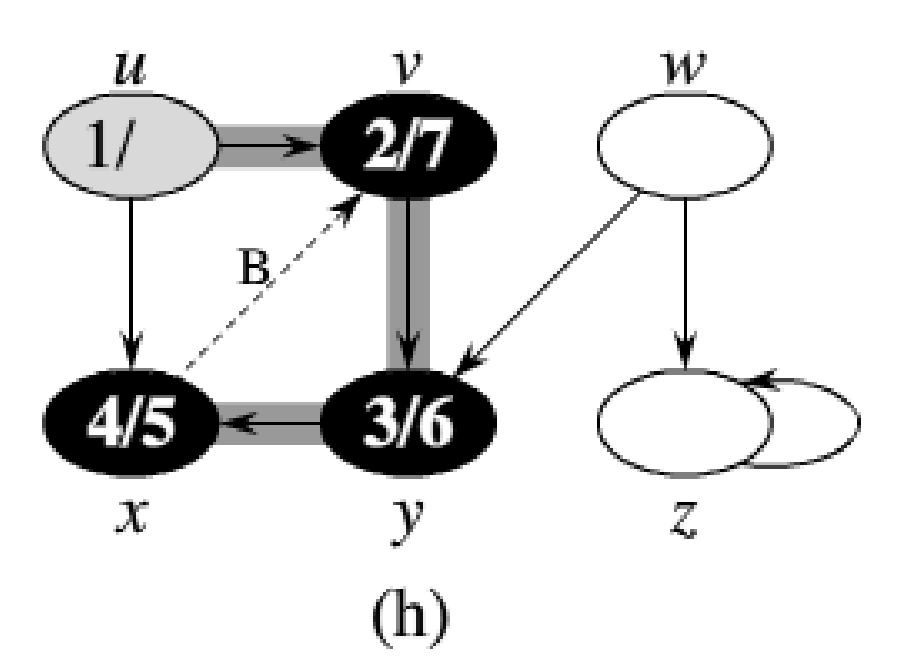
DFS Example:
v Finishes

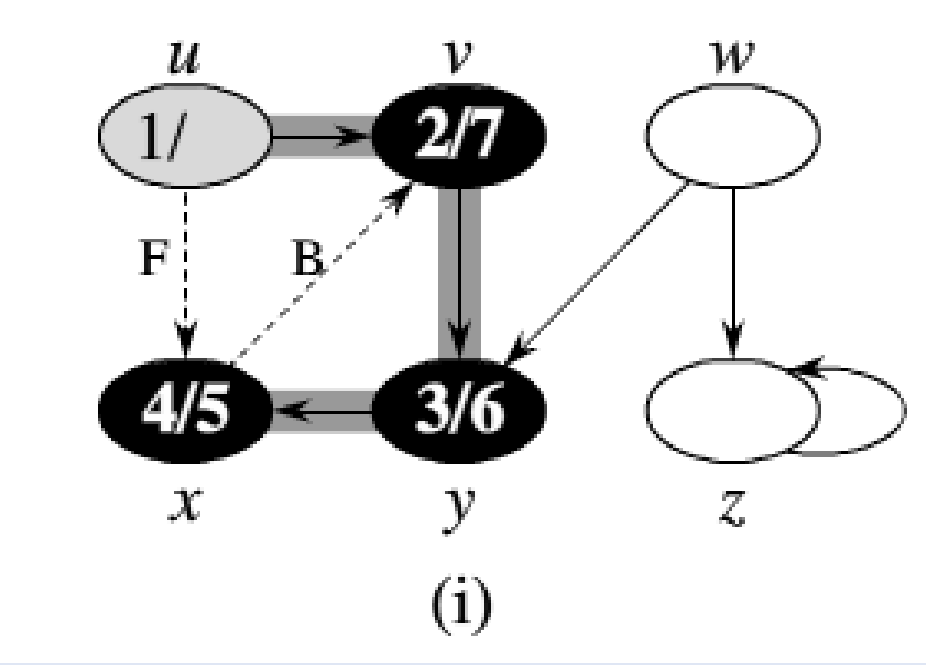




DFS Example: u Discovers Forward Edge

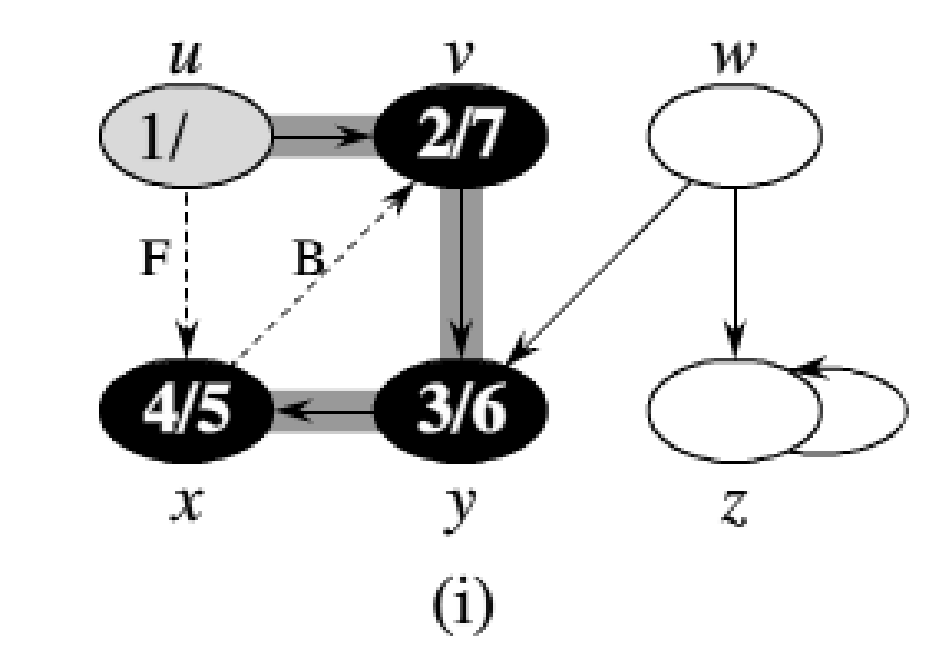
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge

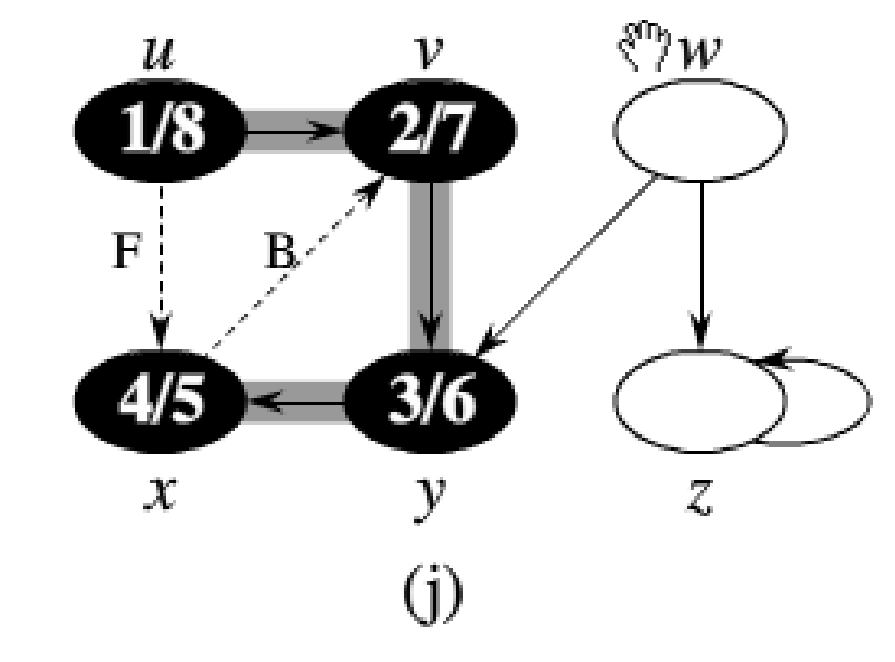




DFS Example: u Finishes

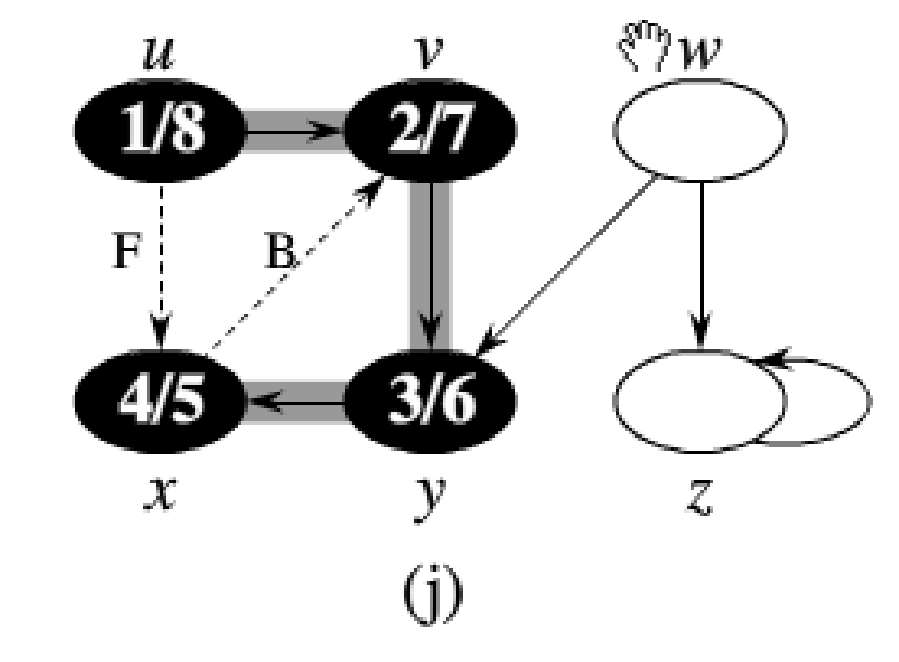
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge

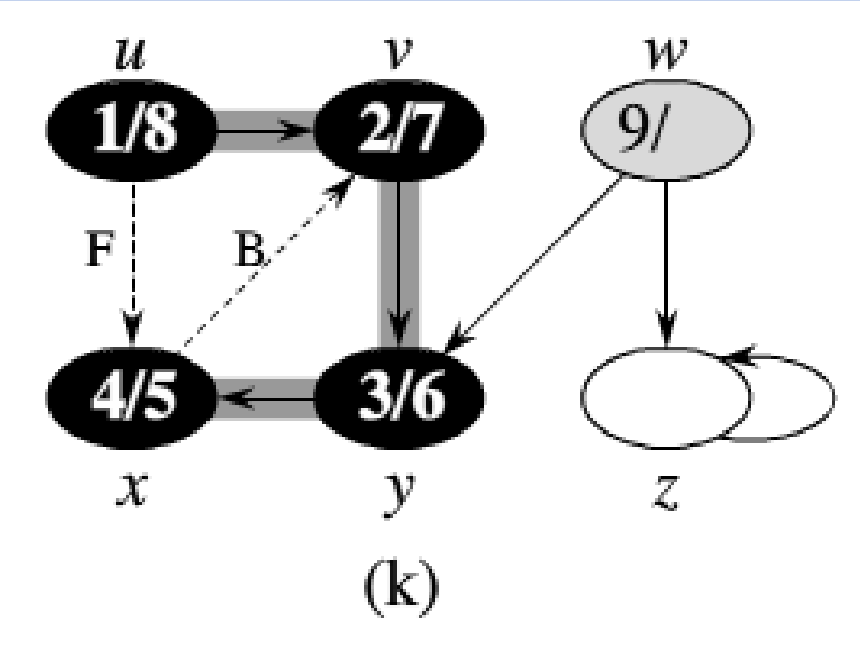




DFS Example: w discovered @ Time = 9

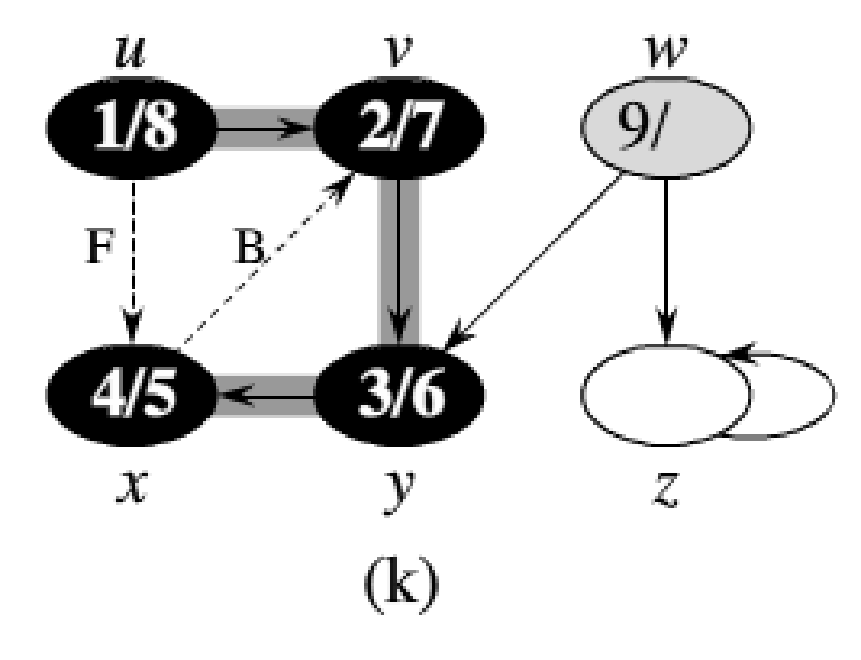
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge // **u.d=1 & x.d=4**
    - u.d > v.d: cross edge

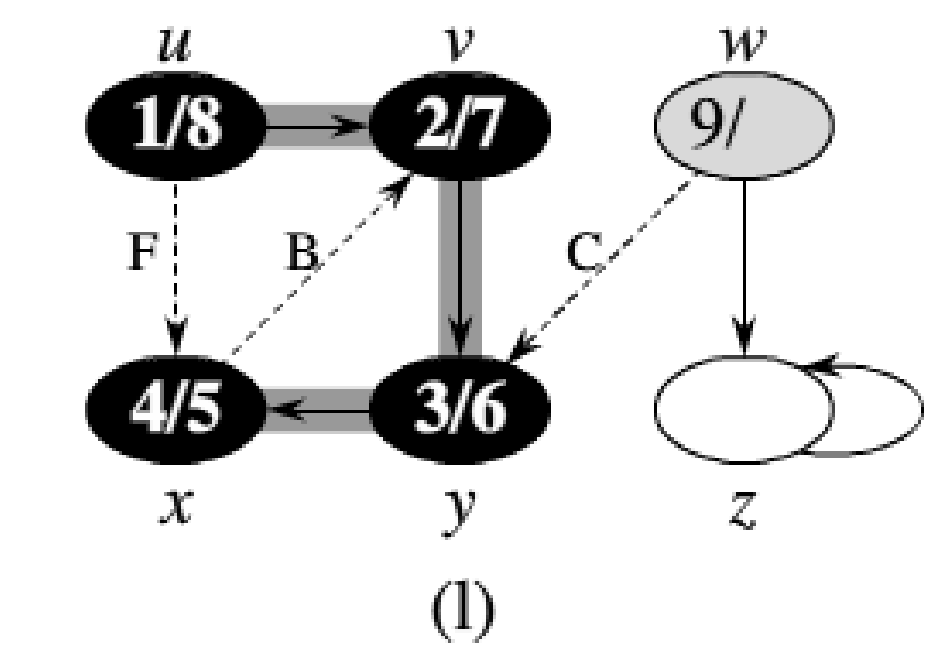




DFS Example: Cross Edge

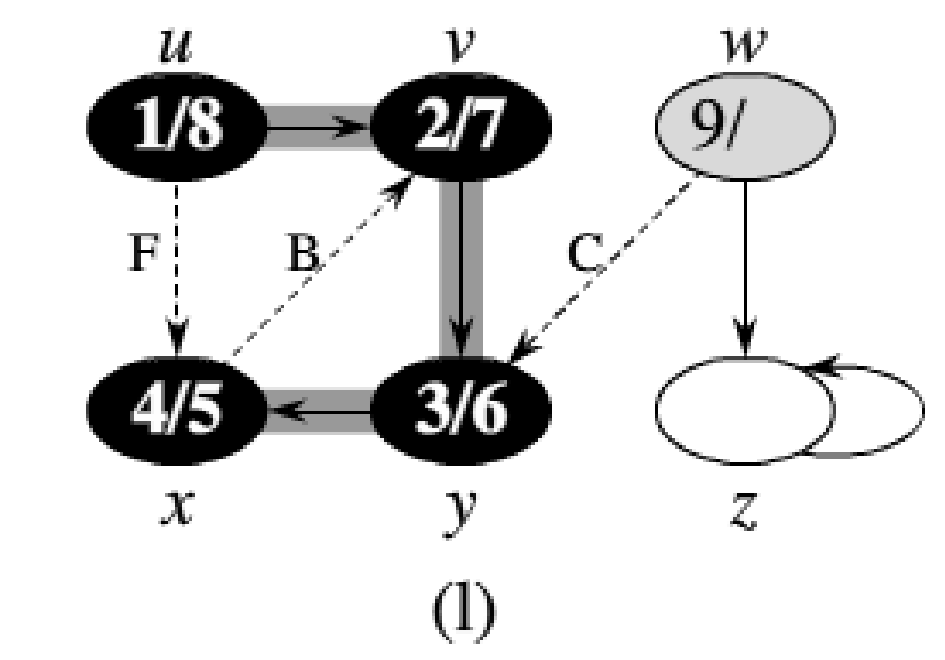
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge **// w.d=9 & y.d=3**

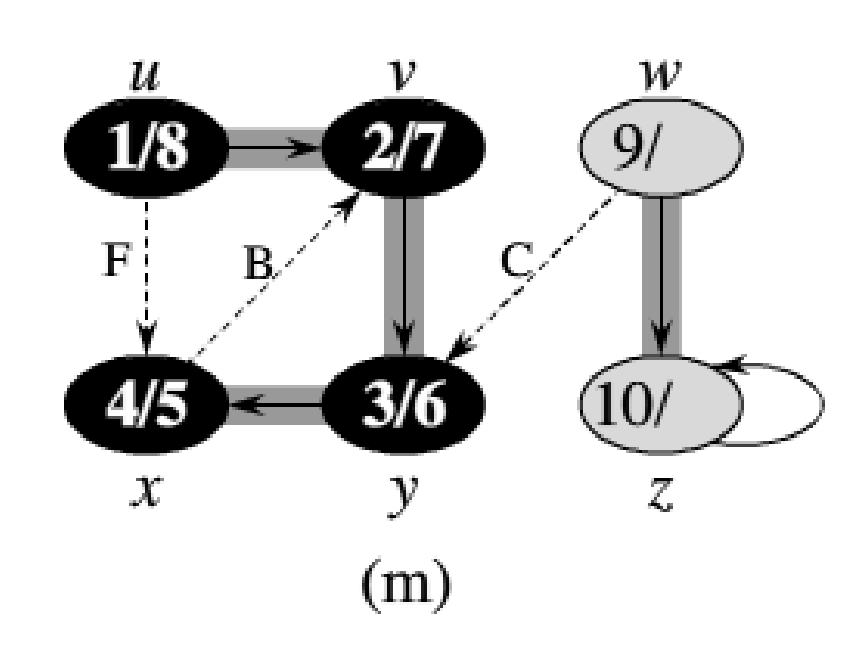




DFS Example:
z discovered @ time=10

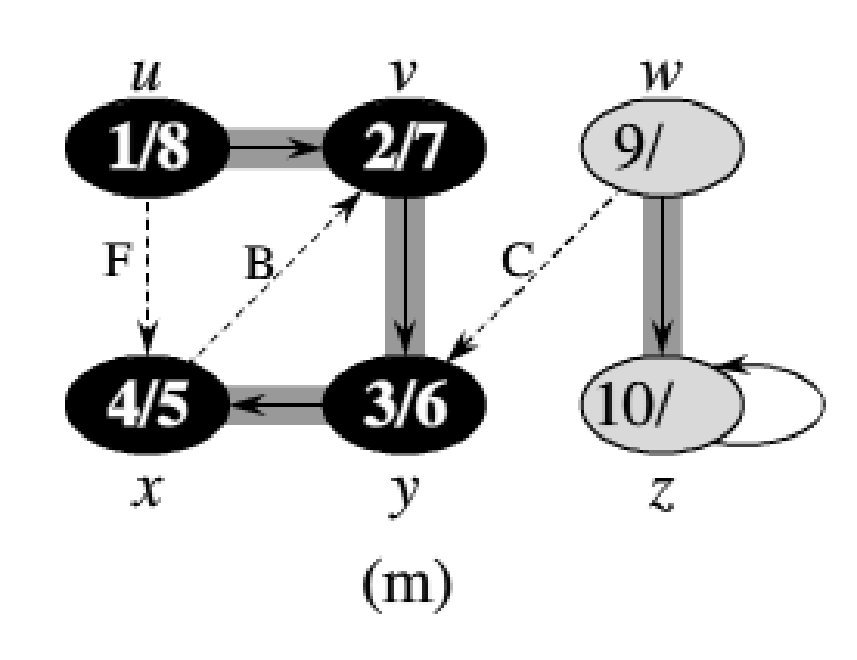
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

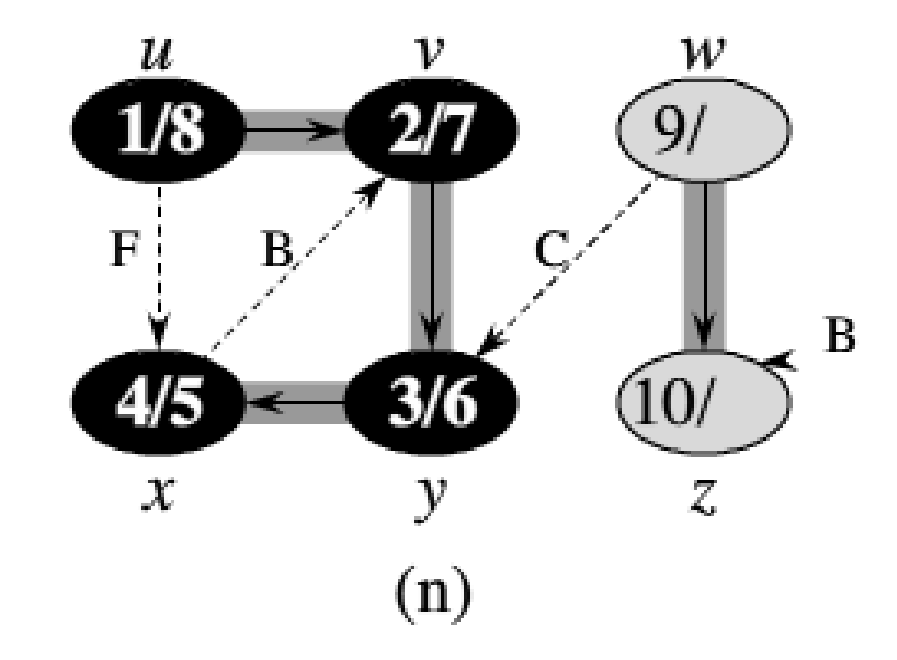




DFS Example: Back Edge

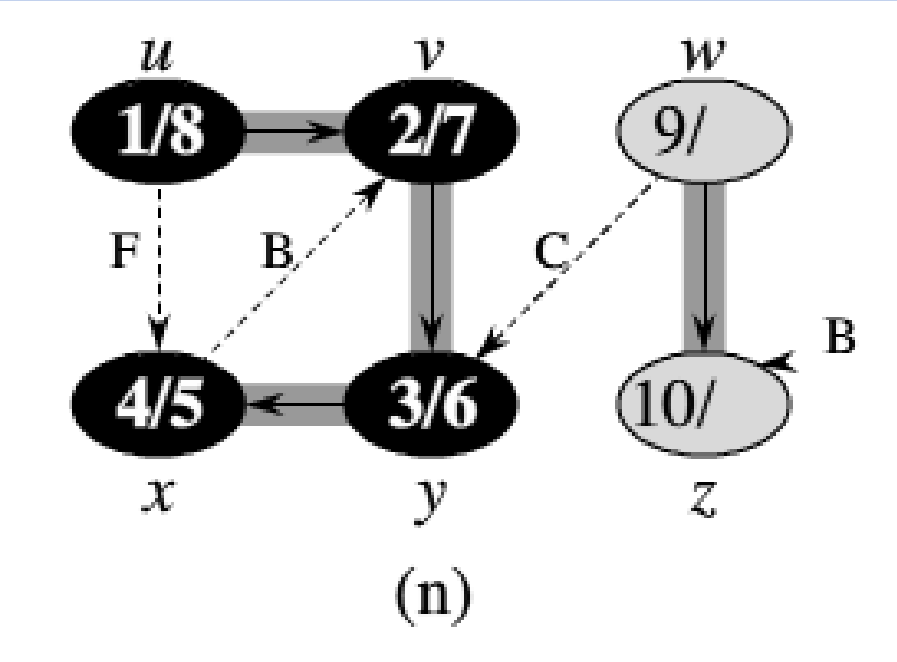
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

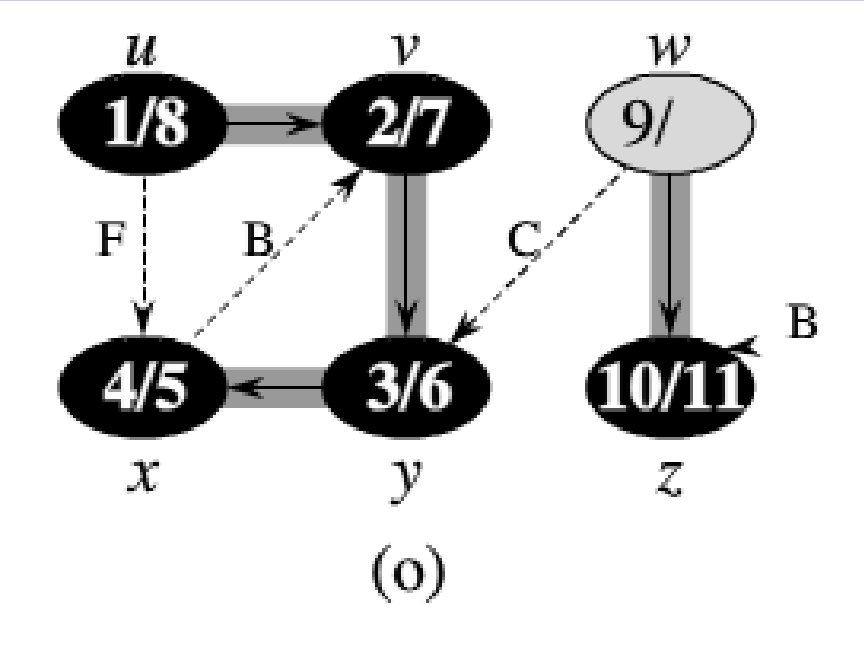




DFS Example: z finishes

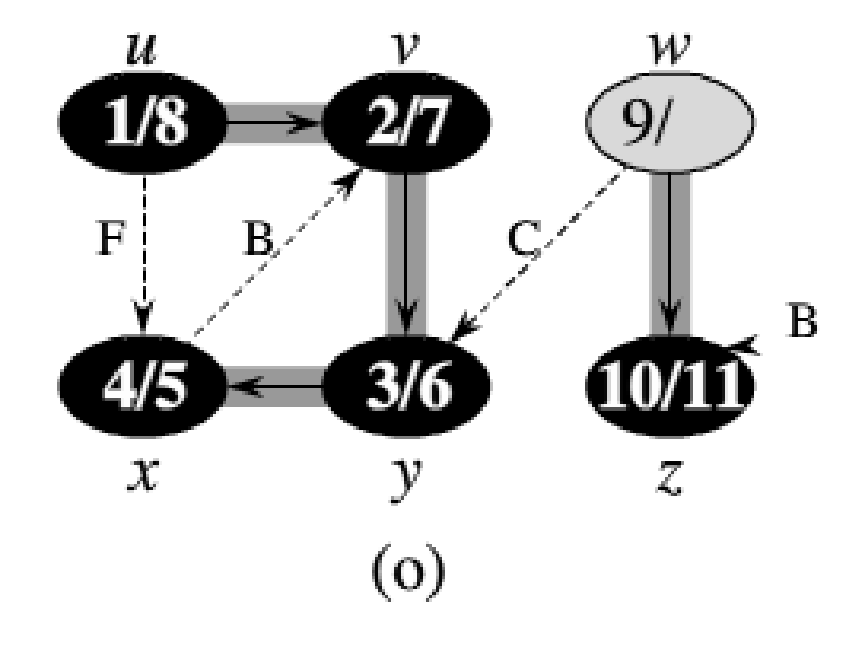
- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

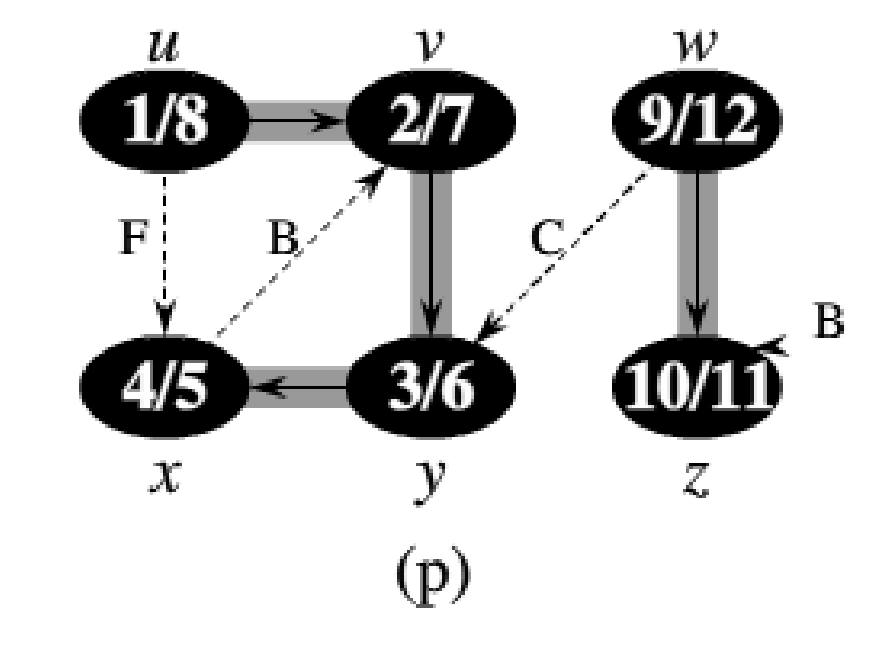




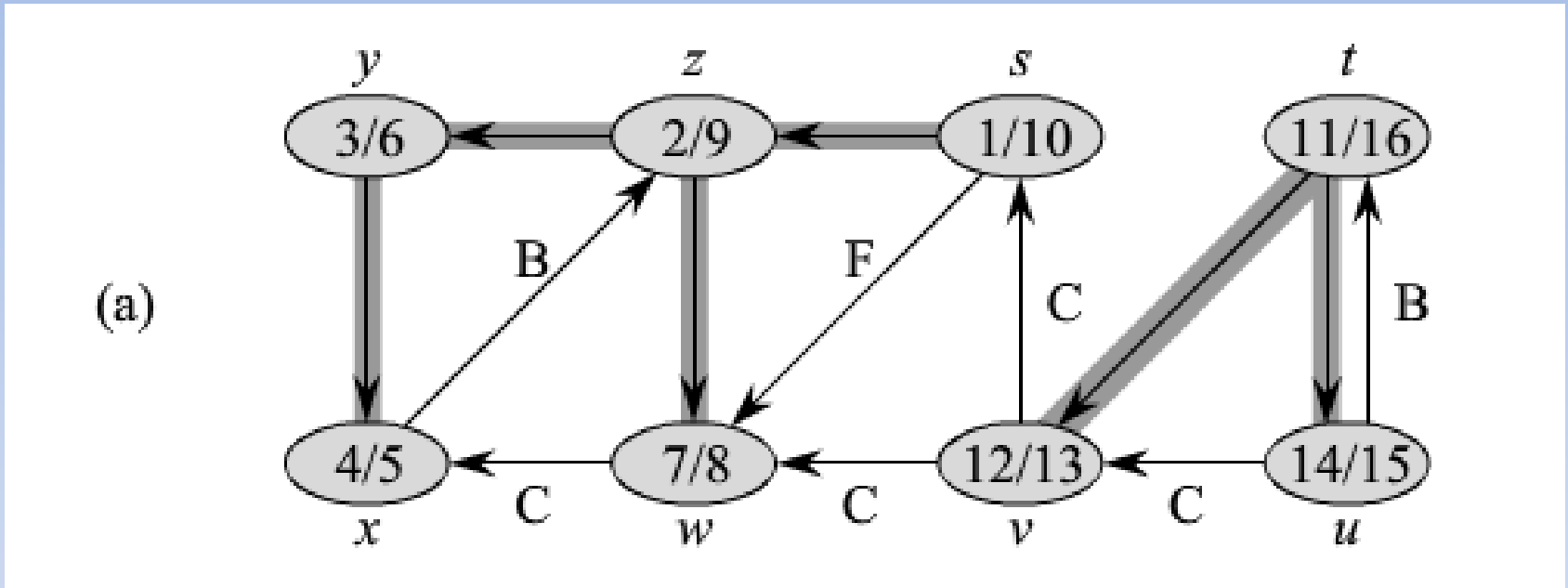
DFS Example: z finishes

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

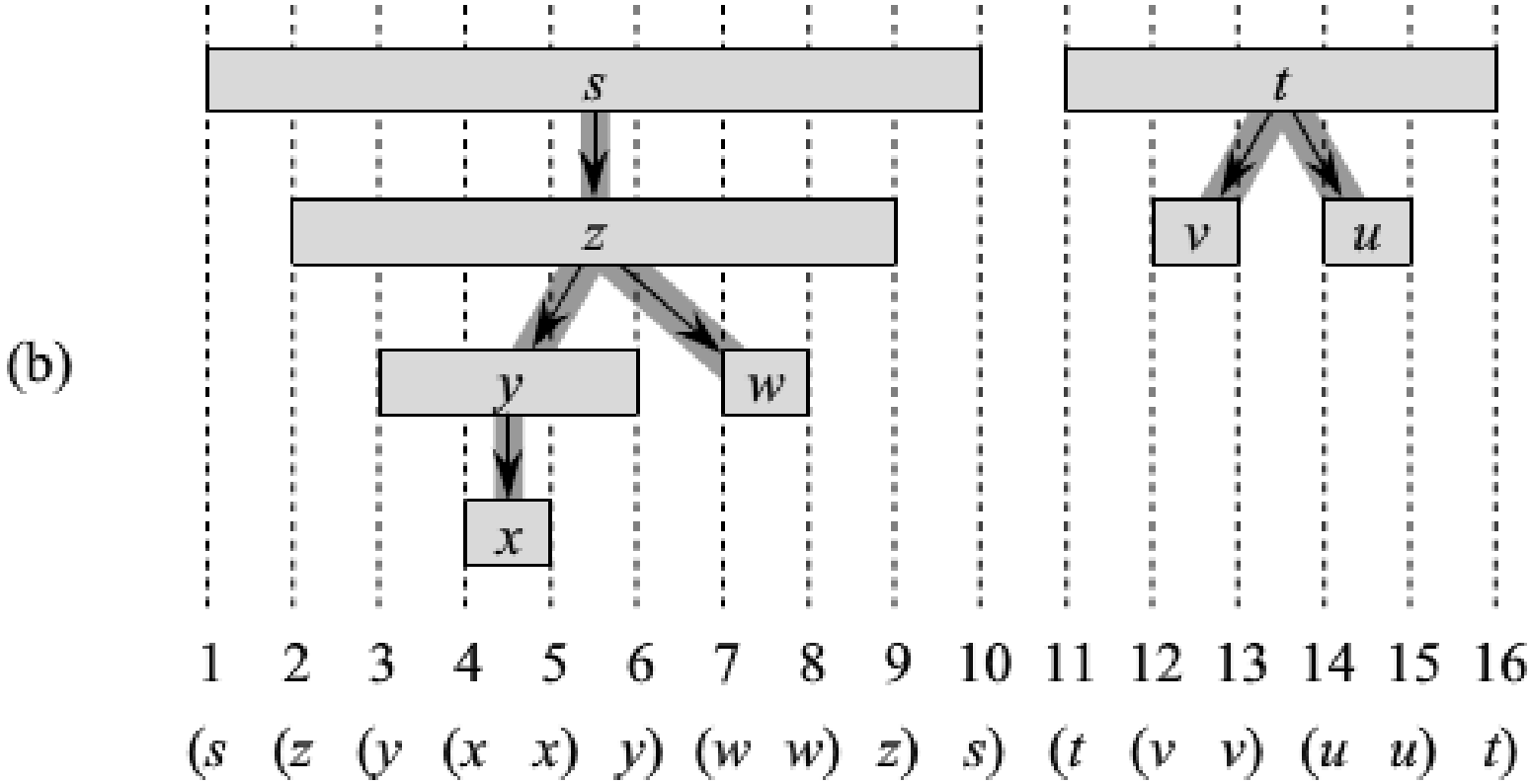




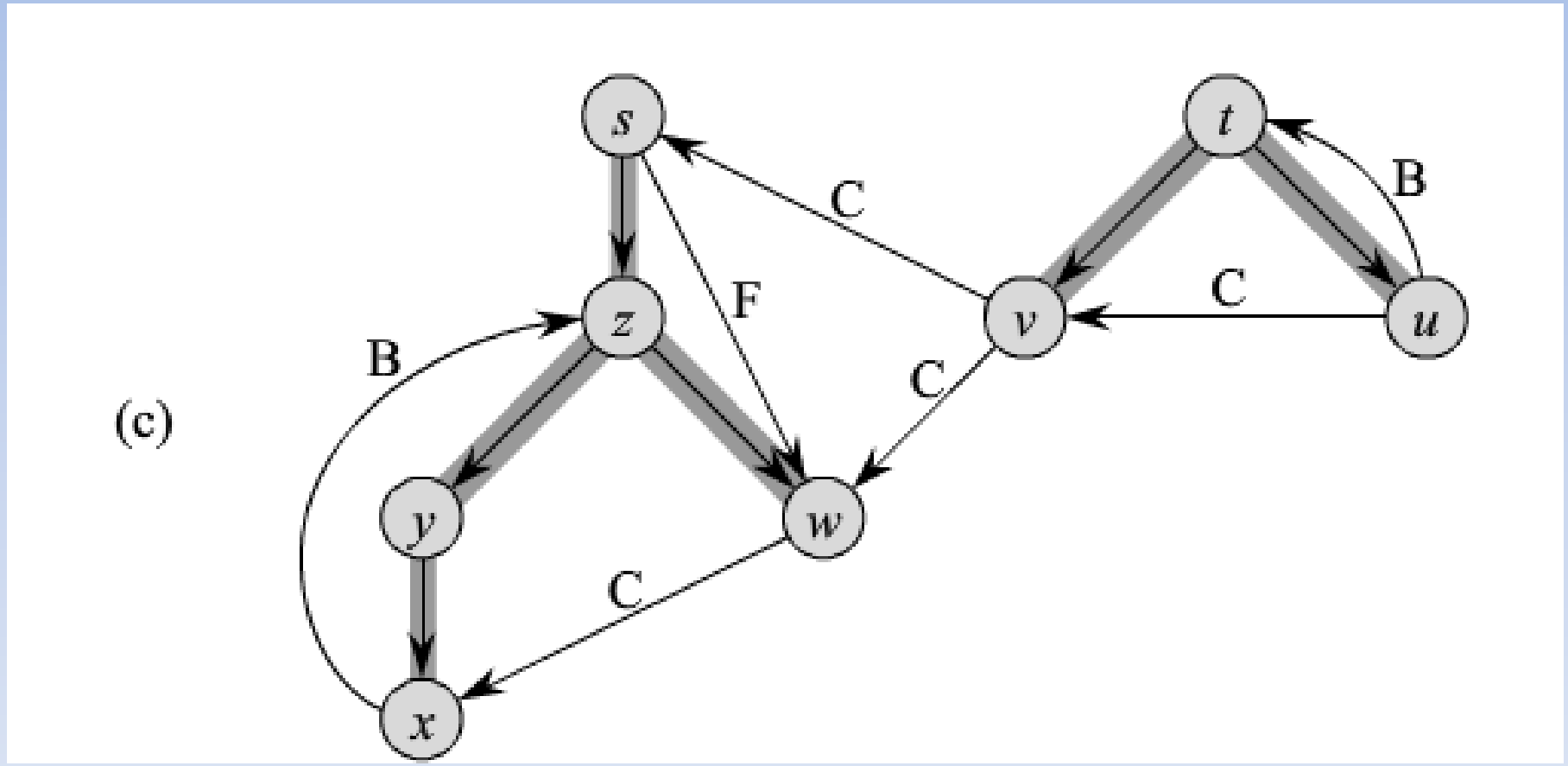
Another DFS Example



Discovery/Finish Times



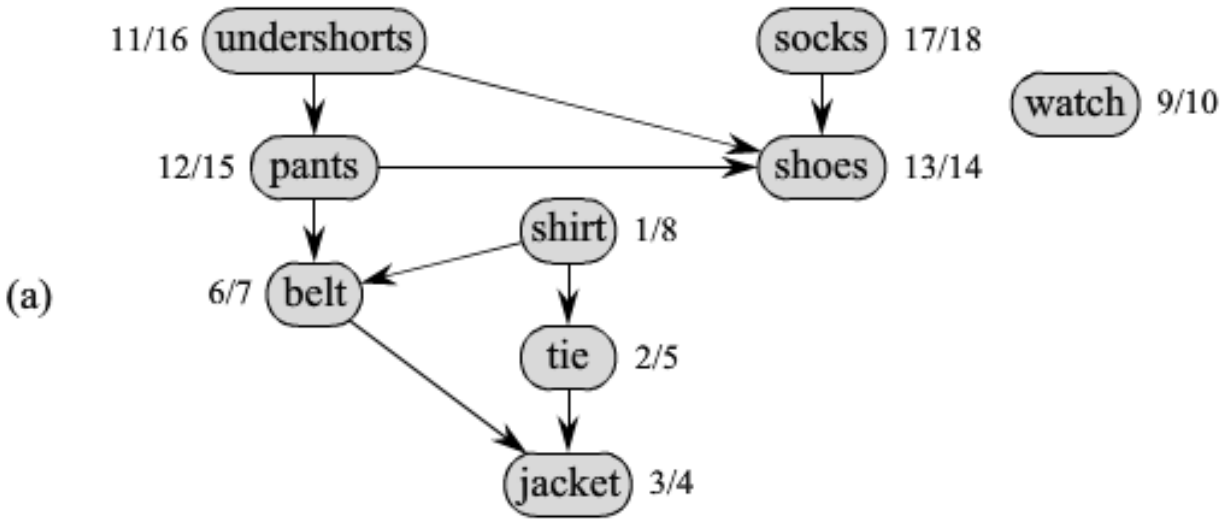
Redrawn Graph

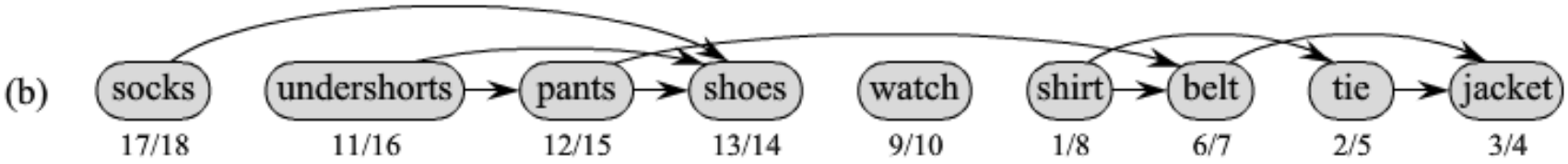


Topological Sort

- Given Directed Acyclic Graph (DAG) $G=(V, E)$
- Linearly order vertices $v \in V$ such that if G contains and edge (u, v) then u appears before v in ordering.

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

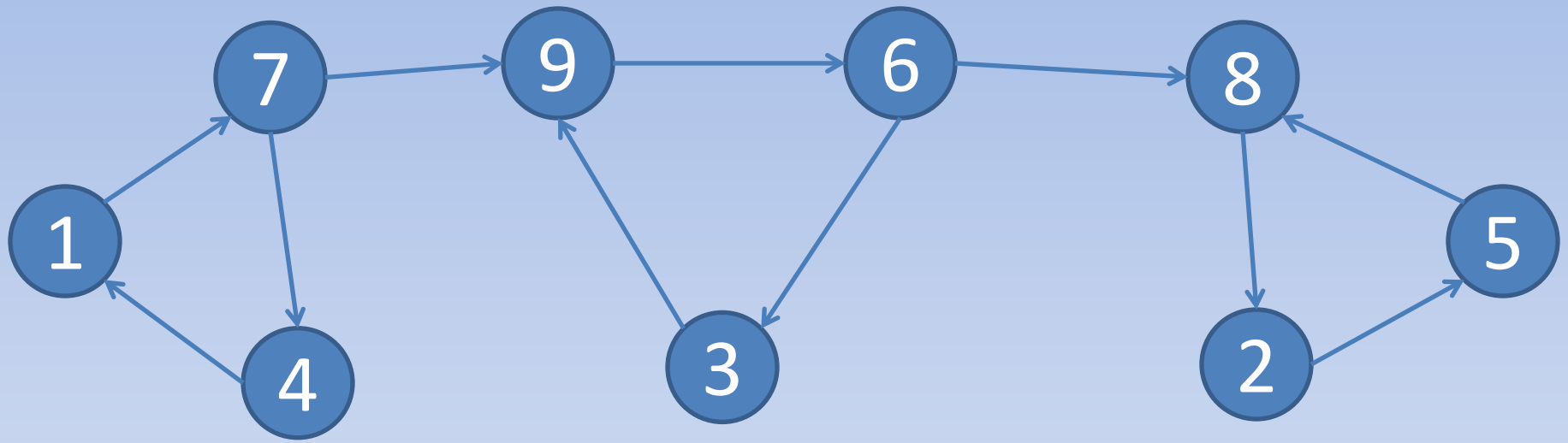

11/16 undershorts
socks 17/18
watch 9/10
12/15 pants
shoes 13/14
shirt 1/8
(a)
6/7 belt
tie 2/5
jacket 3/4


(b)
socks
17/18
undershorts
11/16
pants
12/15
shoes
13/14
watch
9/10
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4

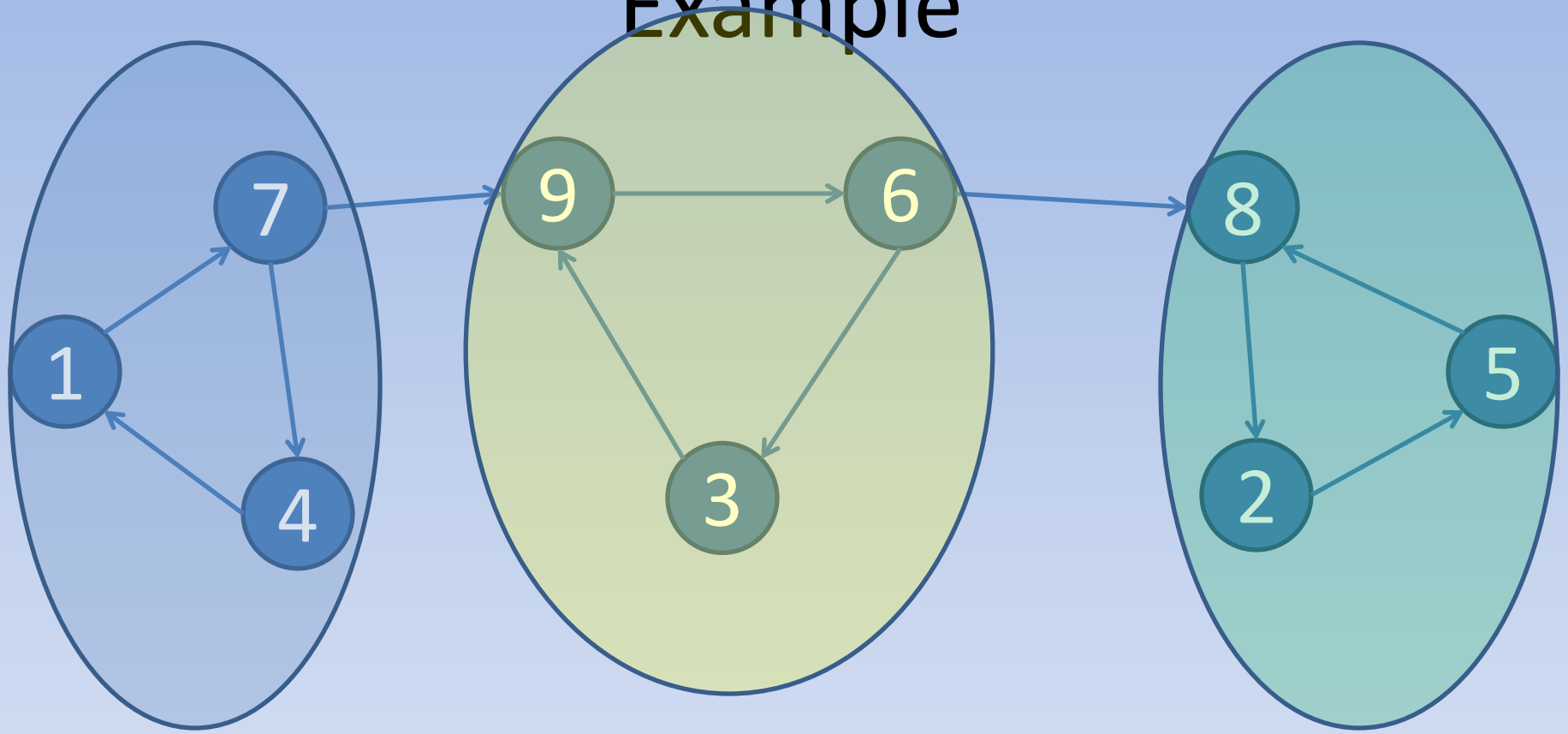
Finding Strongly Connected Components

- Classic Use for DFS!
- We want to decompose a graph into its Strongly Connected Components:

Example

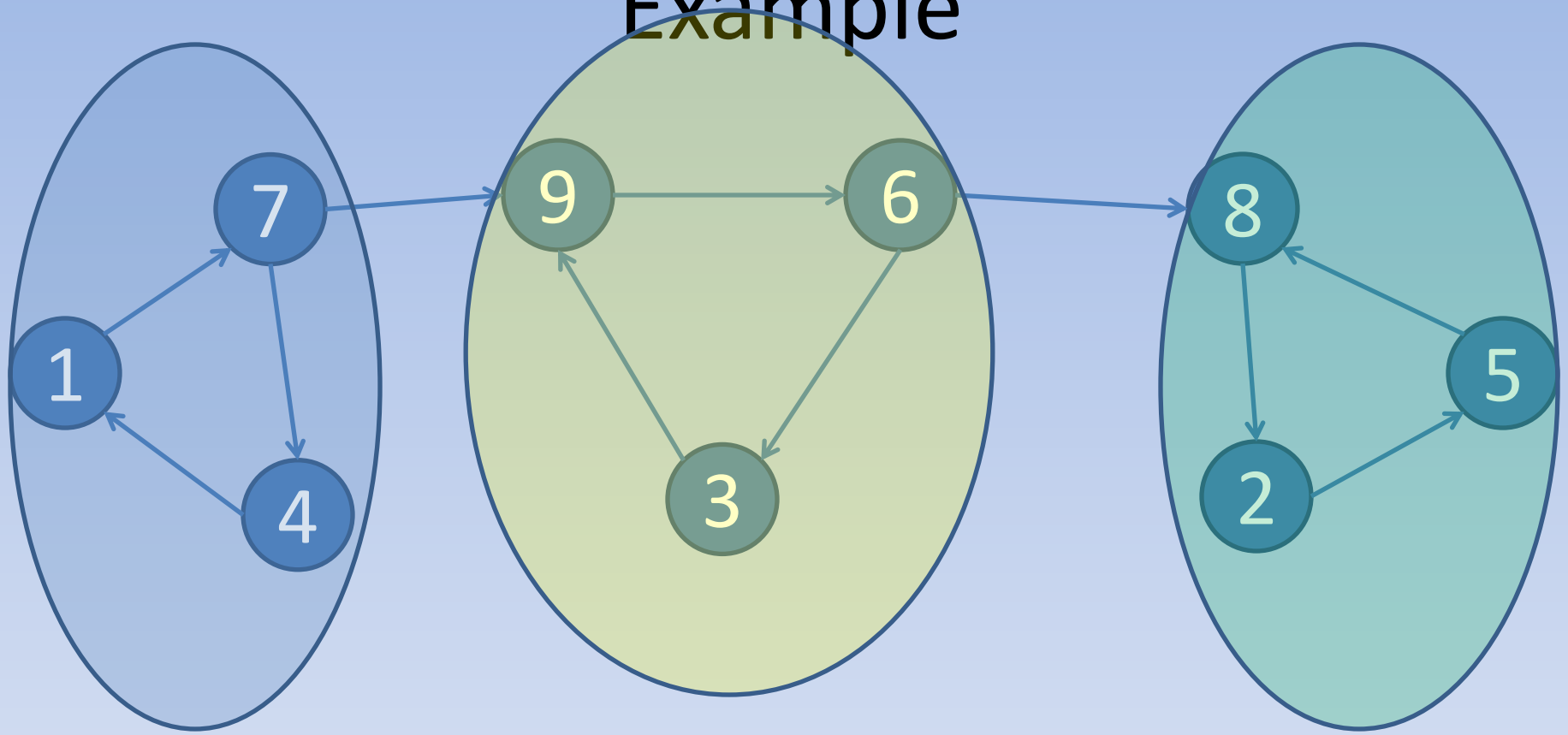


Example



- Every Node is Reachable from Every Other Node w/ Strongly Connected Components

Example

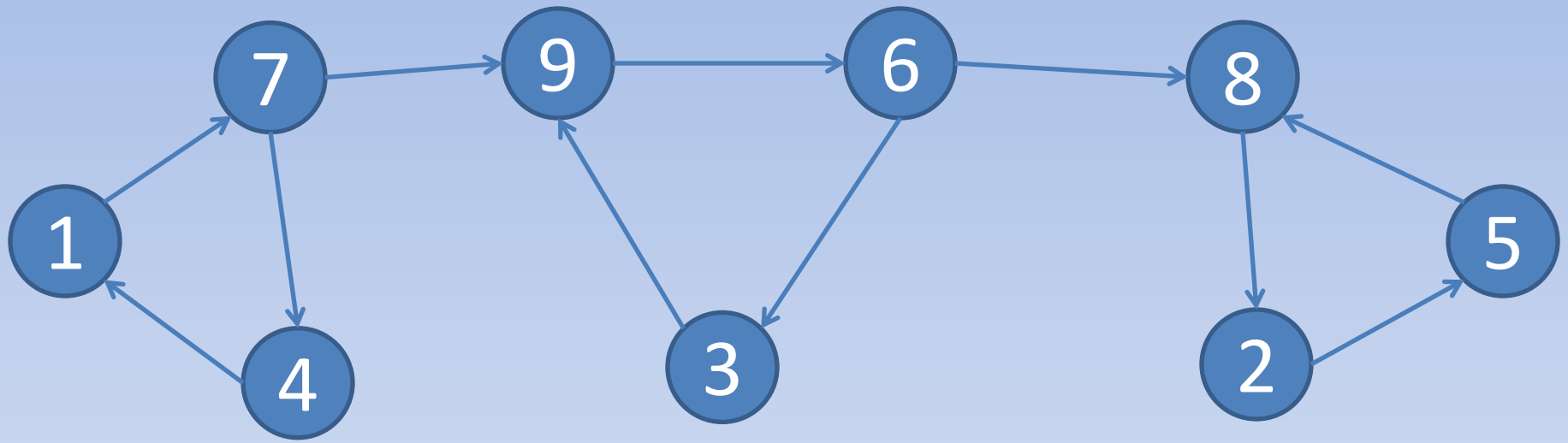


- The maximal set of vertices C ⊆ V such that for every pair of vertices u and v in C, we have both u -->-> v and v -->-> u.

G^T: The Transpose of a Graph

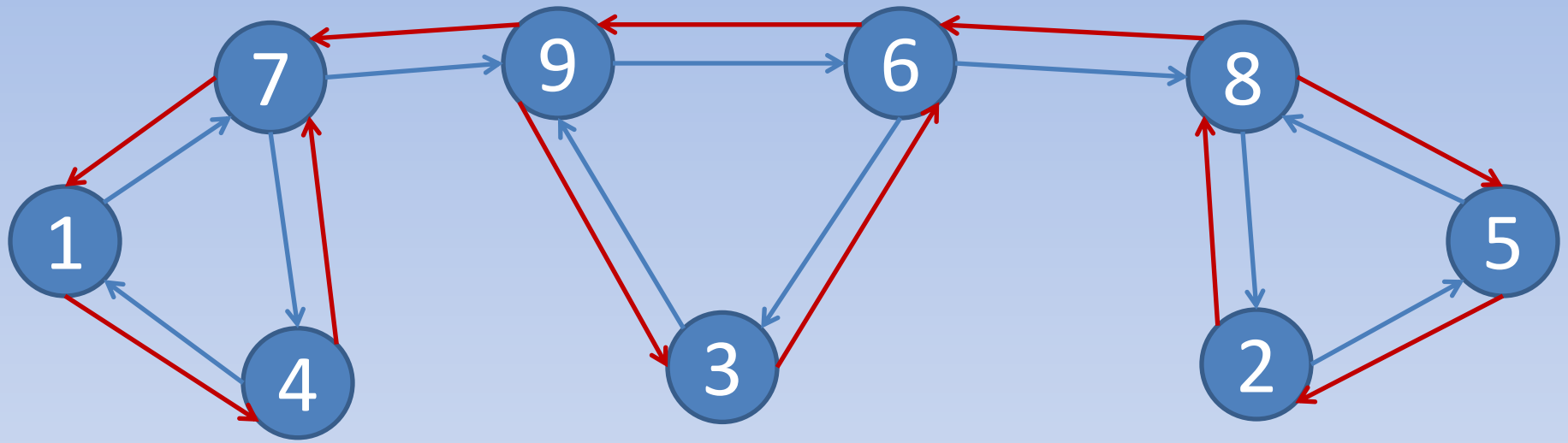
- Our algorithm for finding strongly connected components will utilize the Transpose of a graph.

Example Graph G



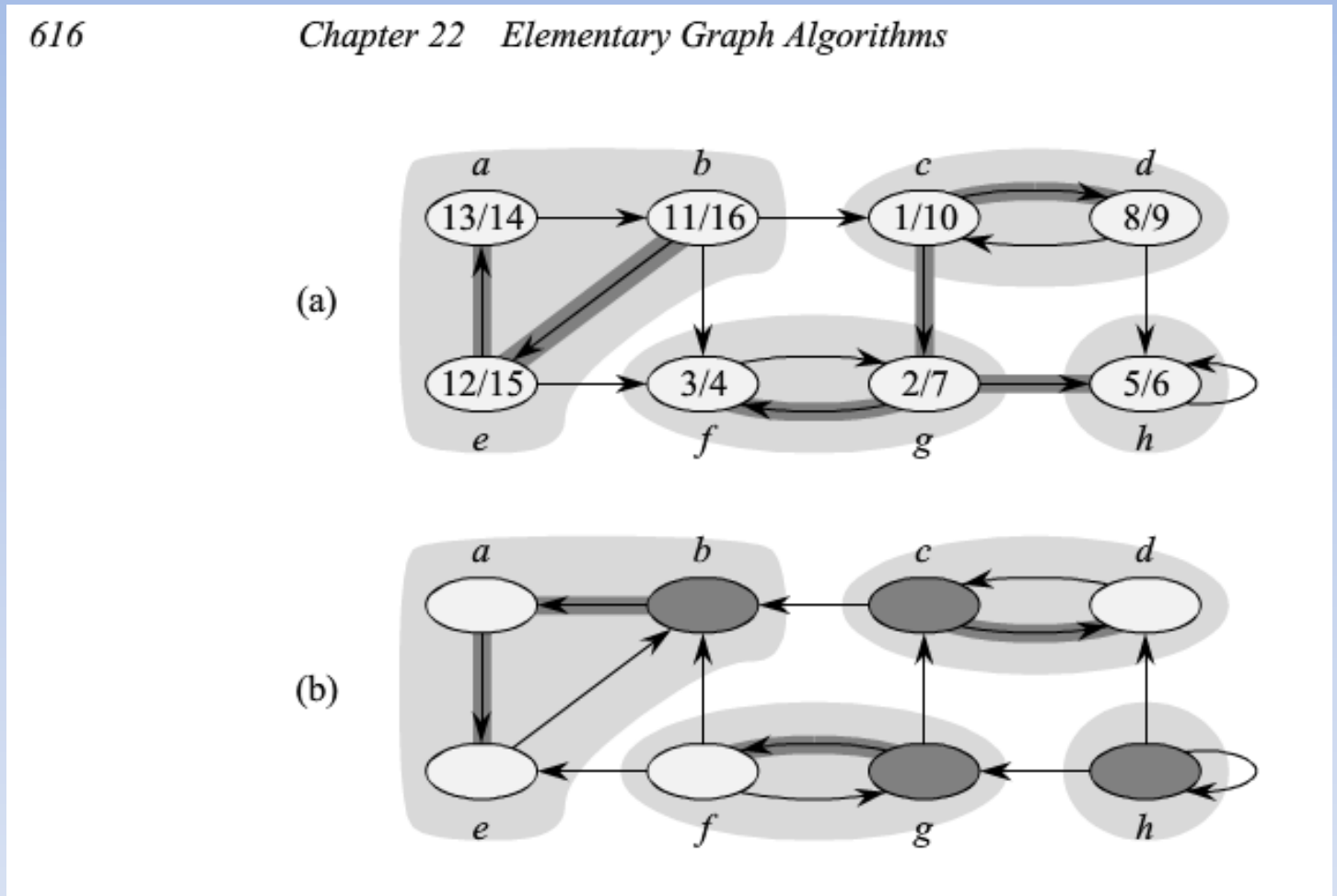
- $G^T = (V, ET)$
  - $E^T = \{u,v): (v,u) \in E\}$

Example Graph G



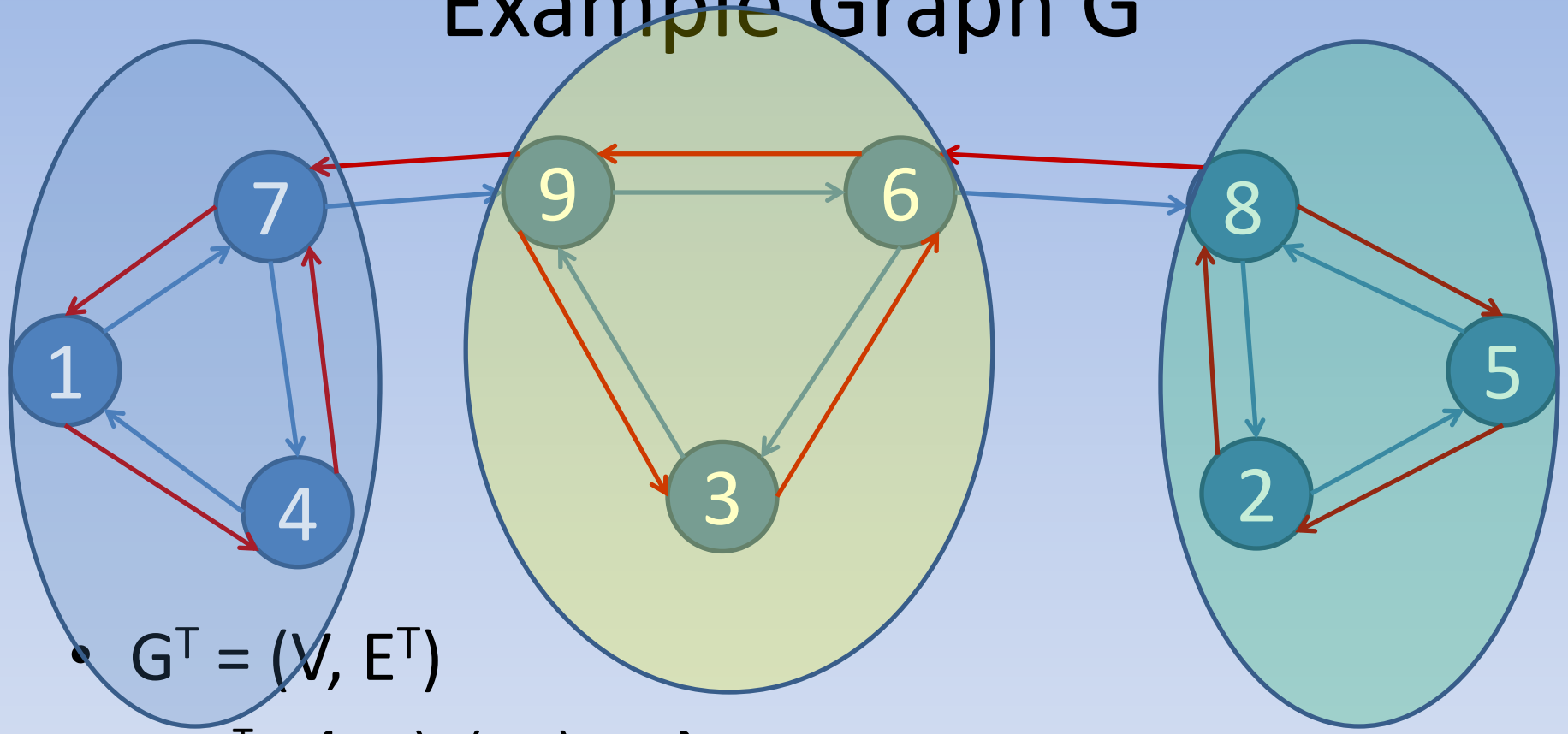
- $G^T = (V, E^T)$
  - $E^T = \{u,v): (v,u) \in E\}$
  - RED Edges!

G and G^t



- G & G^T have SAME connected components!

Example Graph G



- $G^T = (V, E^T)$
  - $E^T = \{u,v): (v,u) \in E\}$
  - RED Edges!
- G & G^T have SAME connected components!

Strongly Connected Components

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

Strongly Connected Components

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

1. RUN DFS to compute finishing times!

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

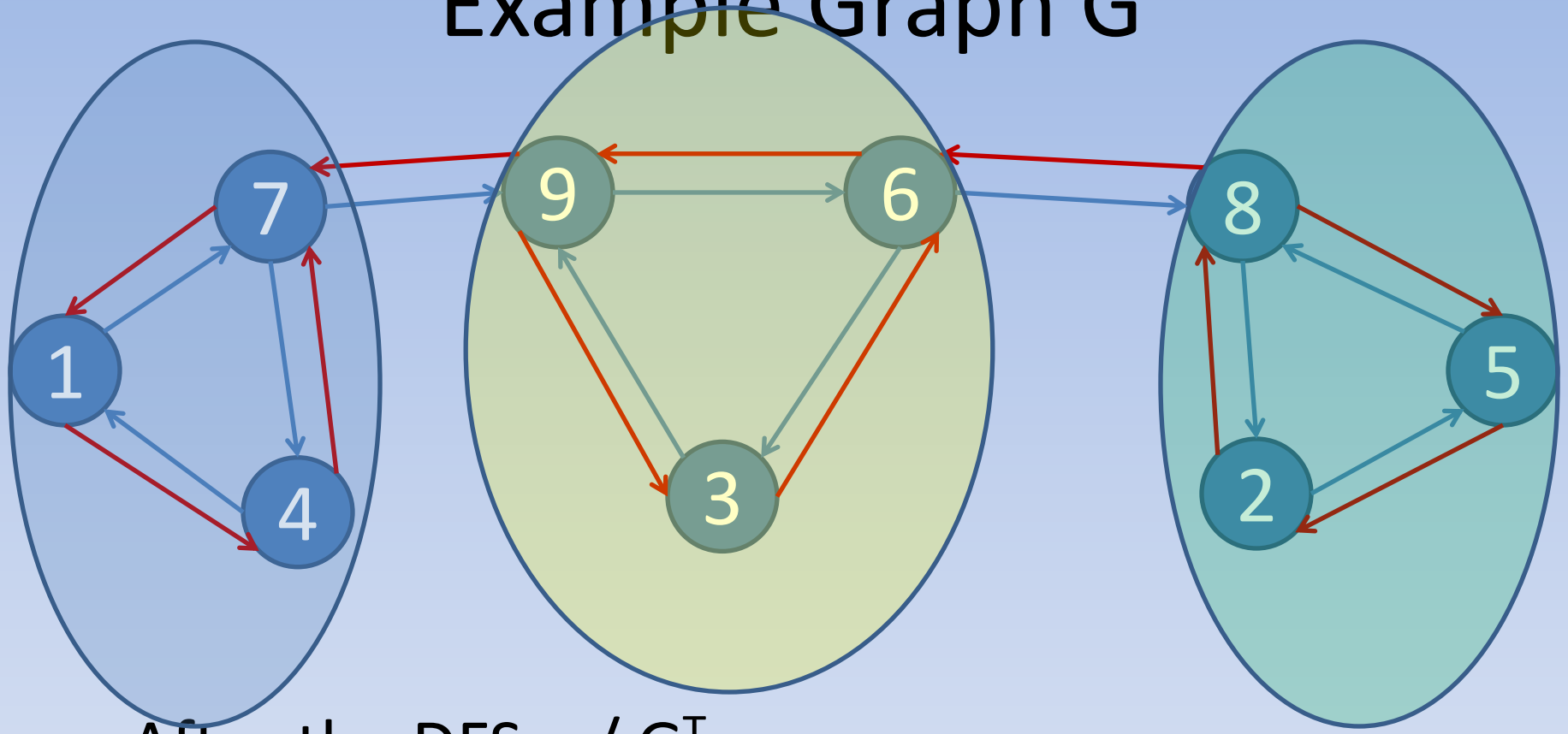
1. RUN DFS to compute finishing times!
2. Compute G^T

Strongly Connected Components

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

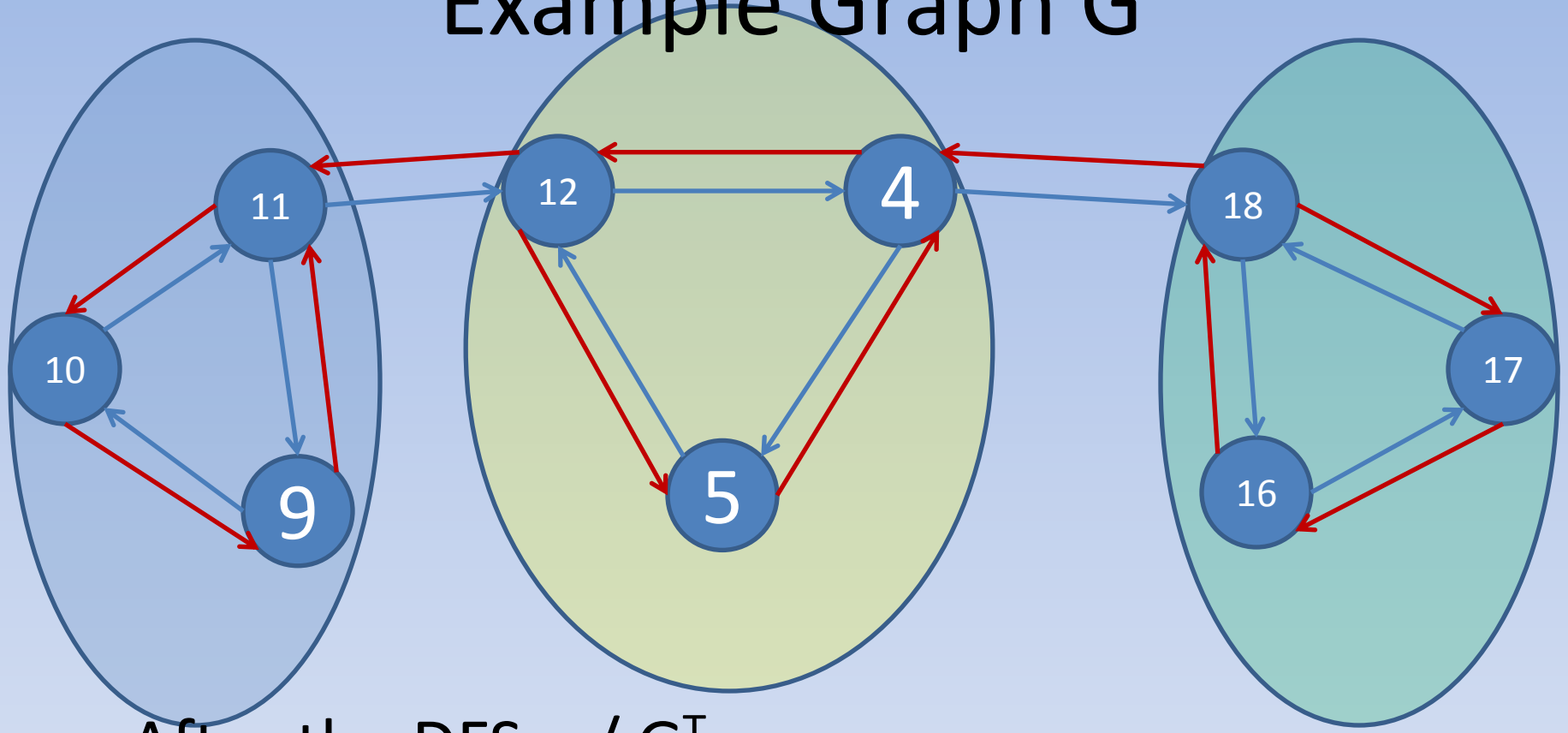
1. RUN DFS to compute finishing times!
2. Compute G^T
3. Run DFS on G^T using finishing time from 1.
4. Out the depth-first tree forest.
   - These will be the strongly connected components.

Example Graph G



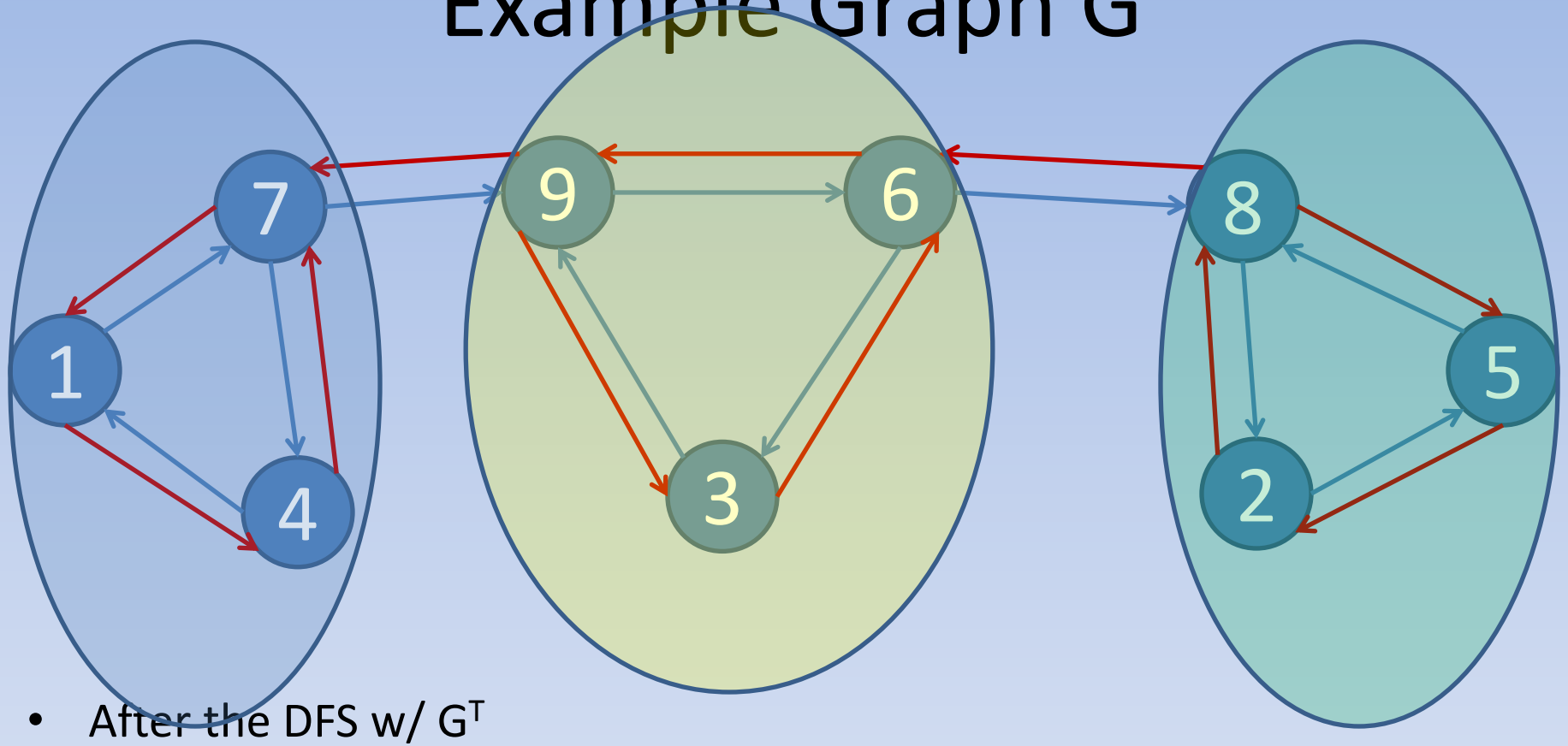
- After the DFS w/ G^T
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]

Example Graph G



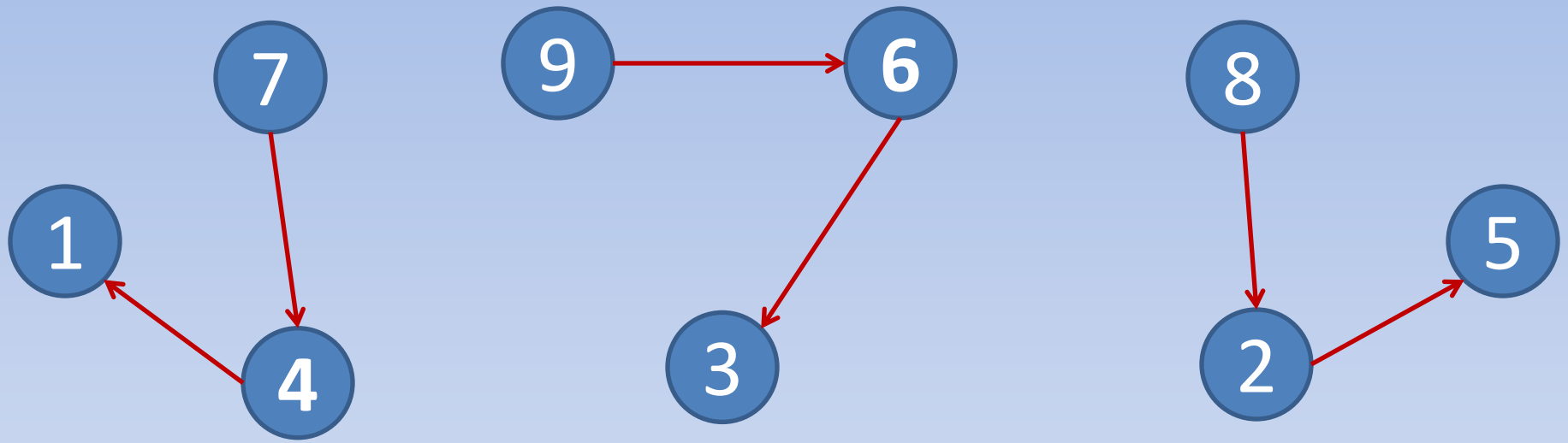
- After the DFS w/ G^T
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]

Example Graph G



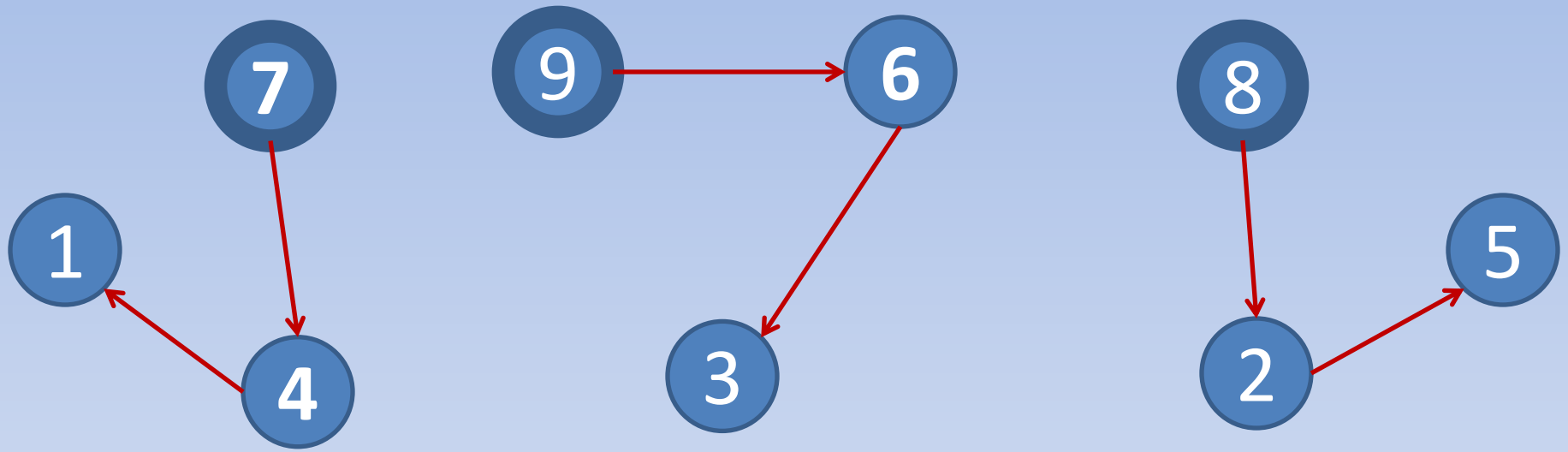
- After the DFS w/ G^T
  - Finish Times = [(4, 6), (5, 3), (9, 4), (10, 1), (11, 7), (12, 9), (16, 2), (17, 5), (18, 8)]
- Second DFS w/ Node Ordering
- Second DFS Vertex Ordering:
  - [8, 5, 2, 9, 7, 1, 4, 3, 6]

Graph G w/ Final Tree Edges



- Final Tree Edges produced by Second DFS

Graph G w/ Final Tree Edges



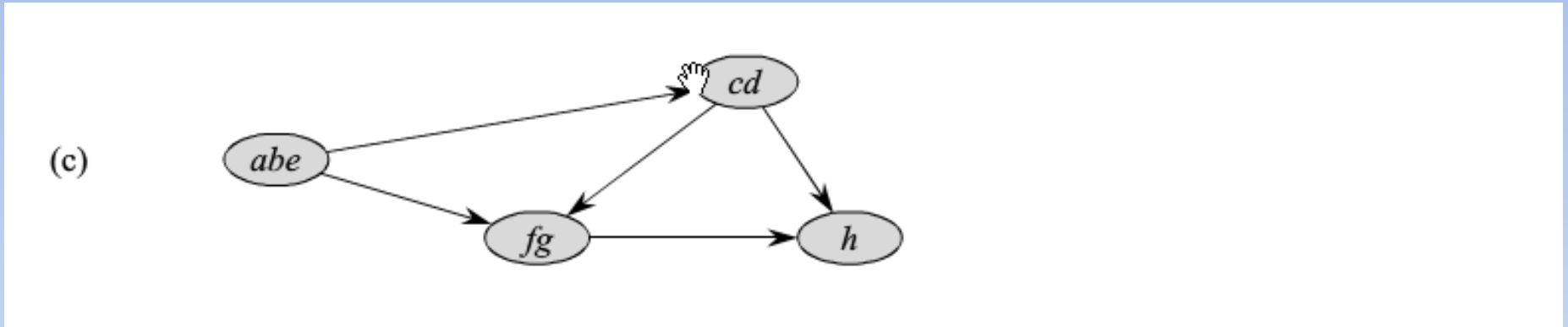
- Final Tree Edges produced by Second DFS

Follow the Leader

- 3 nodes have 7 as leader
- 3 nodes have 8 as leader
- 3 nodes have 9 as leader

- Number of nodes with leader is Size of Clique

Acyclic Component G (G^{SCC})



- Key Idea: Component Graph
  - Constructed contracting edges within strongly connected components.

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

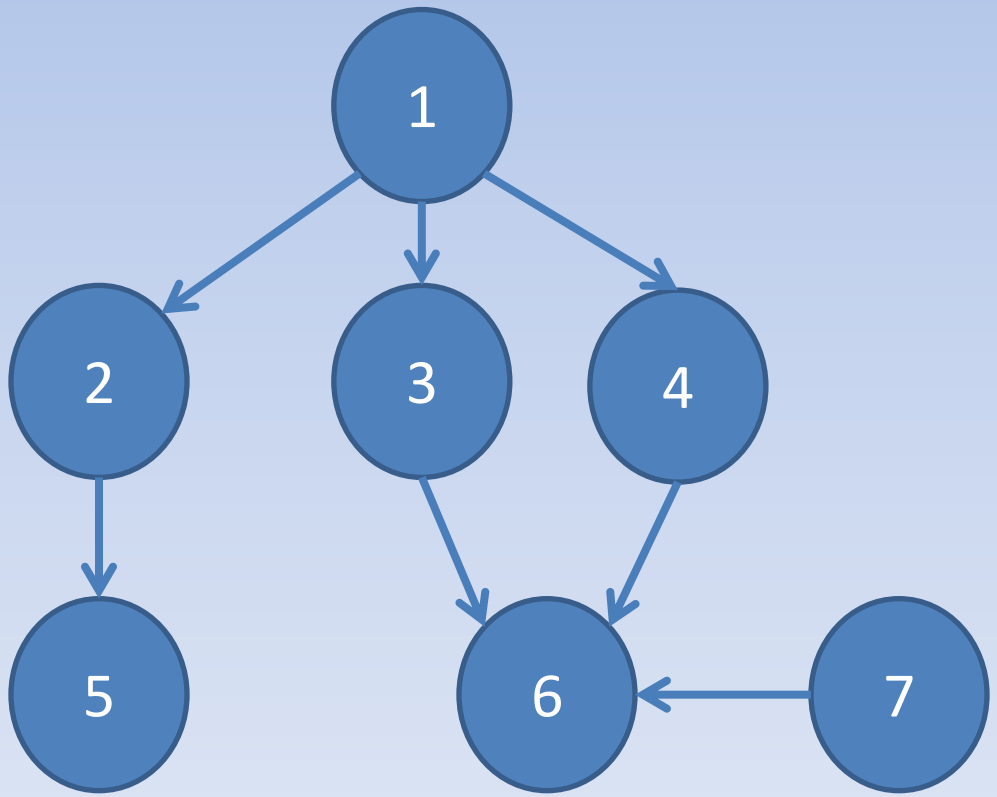
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

(1) Execute BFS(G, 1) and determine the order in which the vertices are explored and the distance from the start vertex 1. Place the information into table below:

Vertex	Order Explored	Distance from 1
1		
2		
3		
4		
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Vertex	Order Explored	Distance from 1
1		
2		
3		
4		
5		
6		
7		

Vertex	Order Explored	Distance from 1
1		
2		
3		
4		
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Order Explored	Distance from 1
1	1	0
2		
3		
4		
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Order Explored	Distance from 1
1	1	0
2	2	1
3		
4		
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Order Explored	Distance from 1
1	1	0
2	2	1
3	3	1
4		
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Order Explored	Distance from 1
1	1	0
2	2	1
3	3	1
4	4	1
5		
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Order Explored	Distance from 1
1	1	0
2	2	1
3	3	1
4	4	1
5	5	2
6		
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

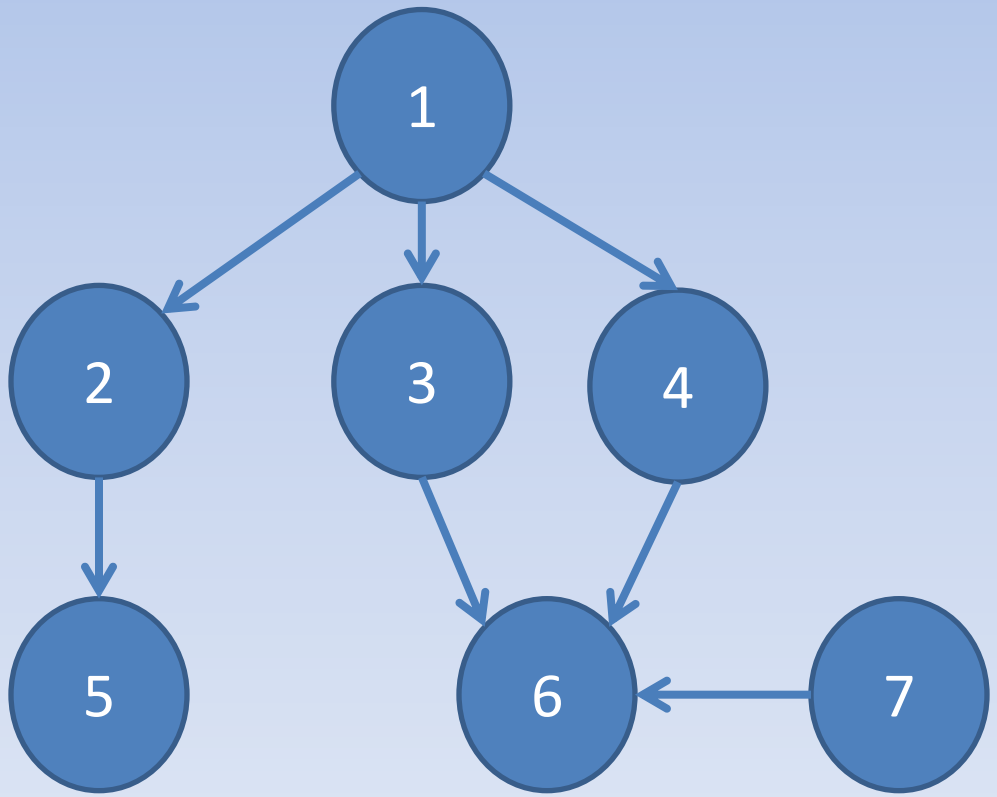
Vertex	Order Explored	Distance from 1
1	1	0
2	2	1
3	3	1
4	4	1
5	5	2
6	6	2
7		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

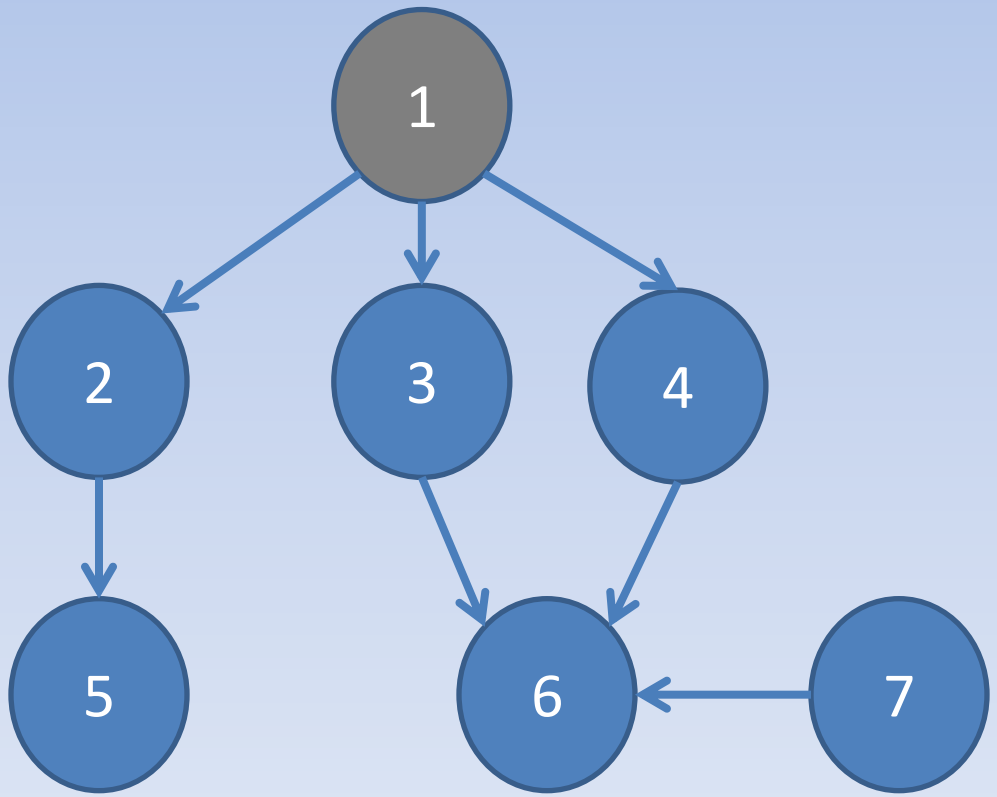
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



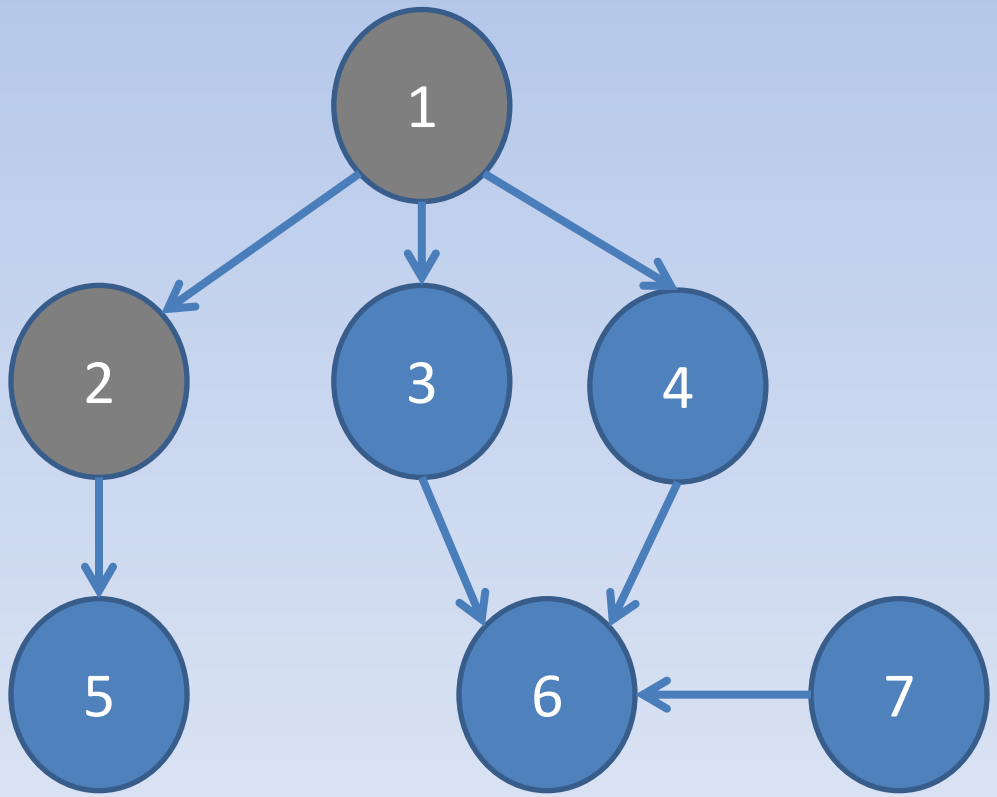
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2			
3			
4			
5			
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



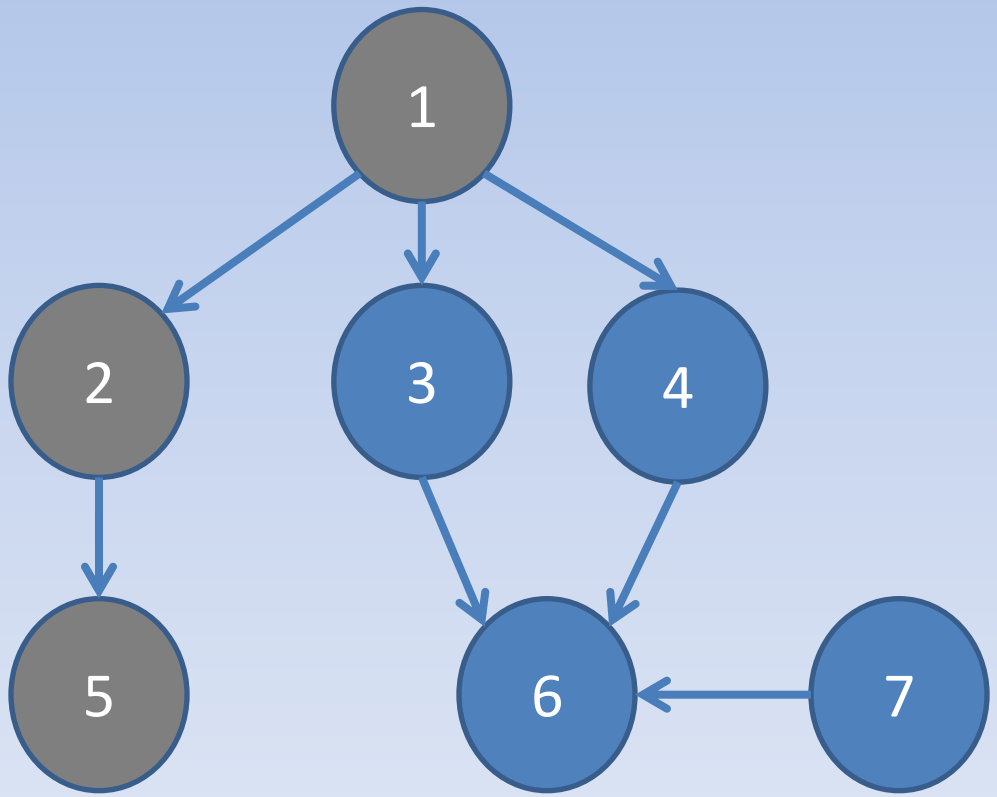
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2		
3			
4			
5			
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



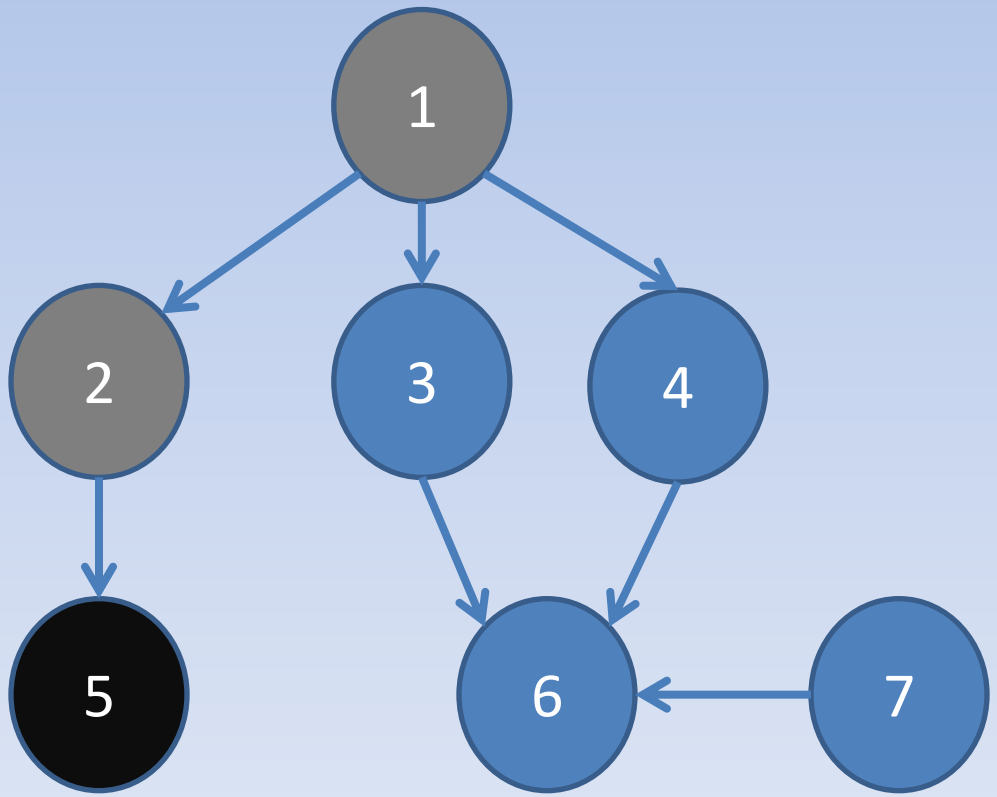
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2		
3			
4			
5	3		
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



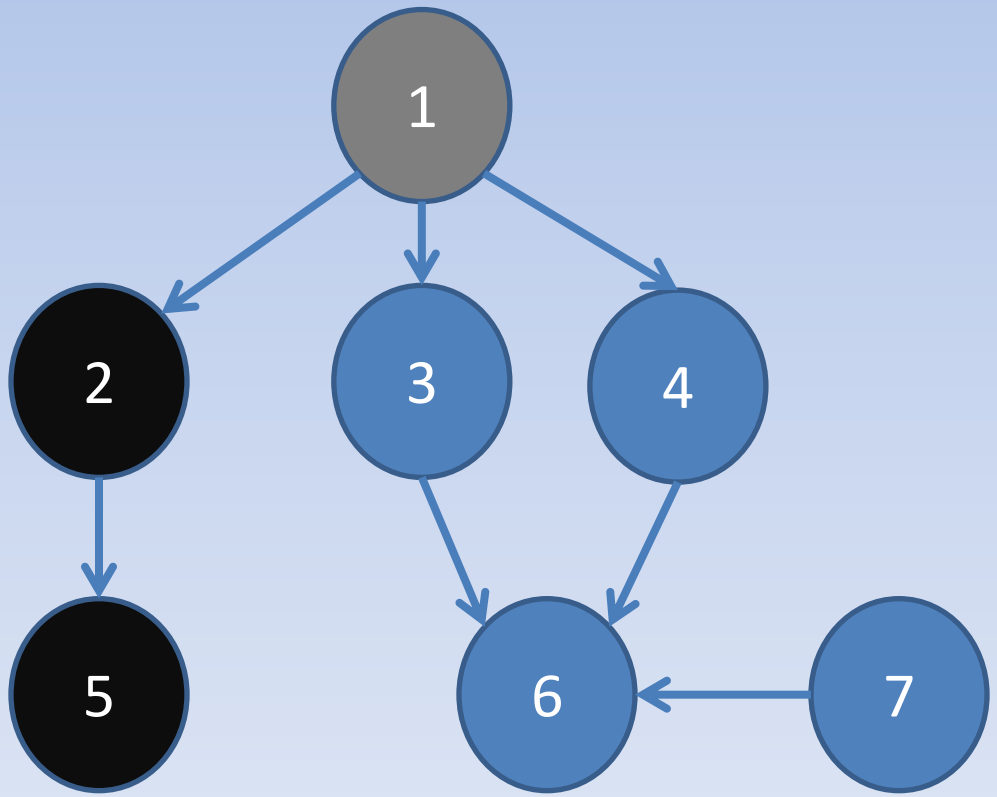
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2		
3			
4			
5	3	4	
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



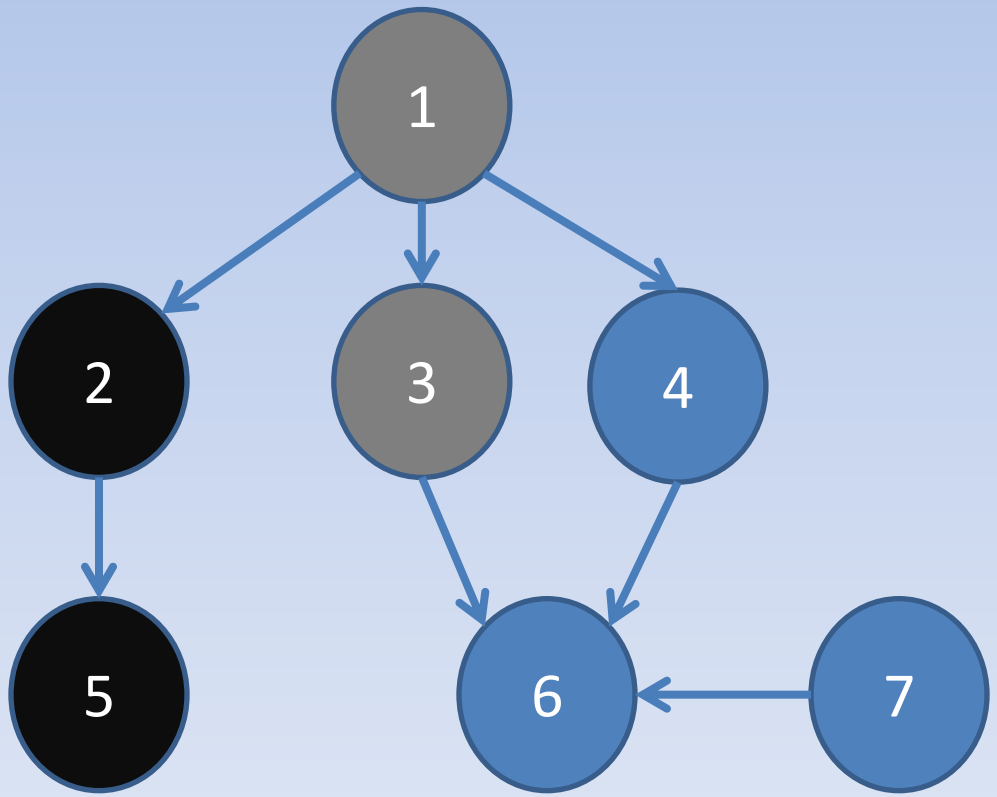
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3			
4			
5	3	4	
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



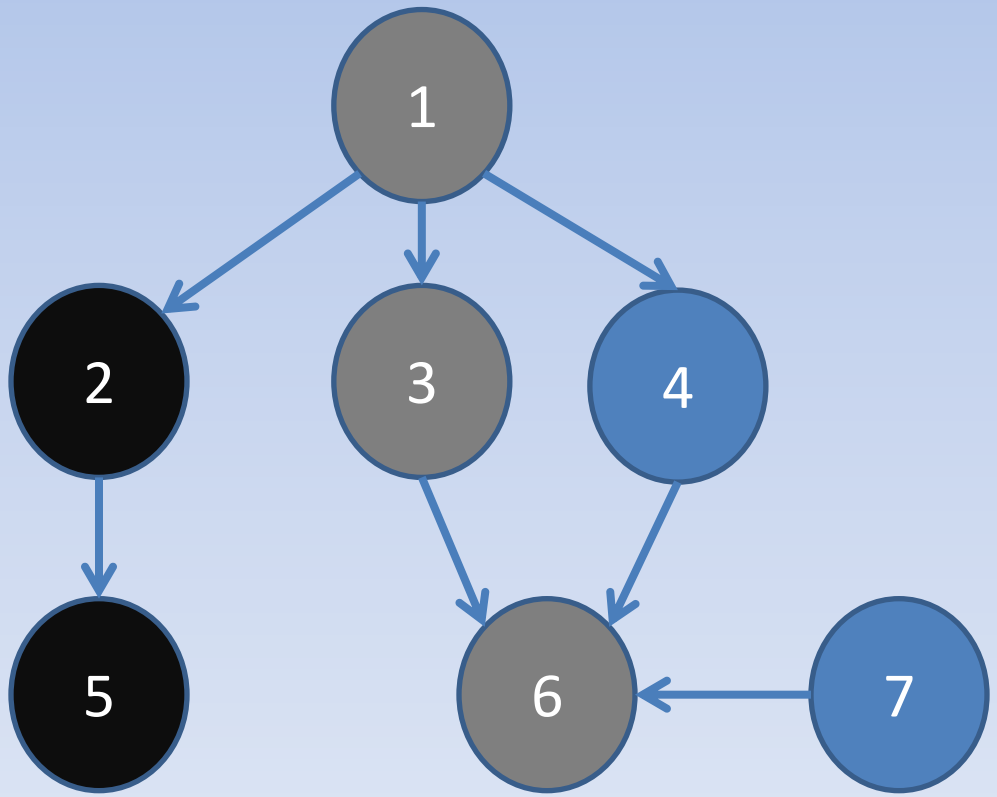
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6		
4			
5	3	4	
6			
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



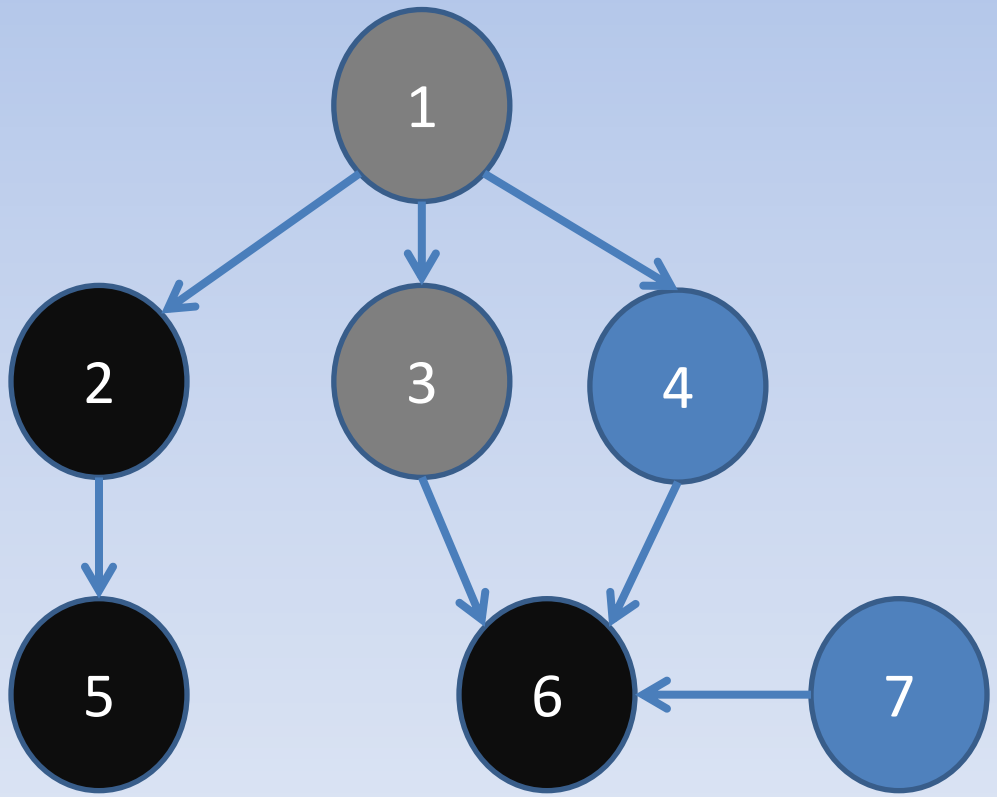
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6		
4			
5	3	4	
6	7		
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



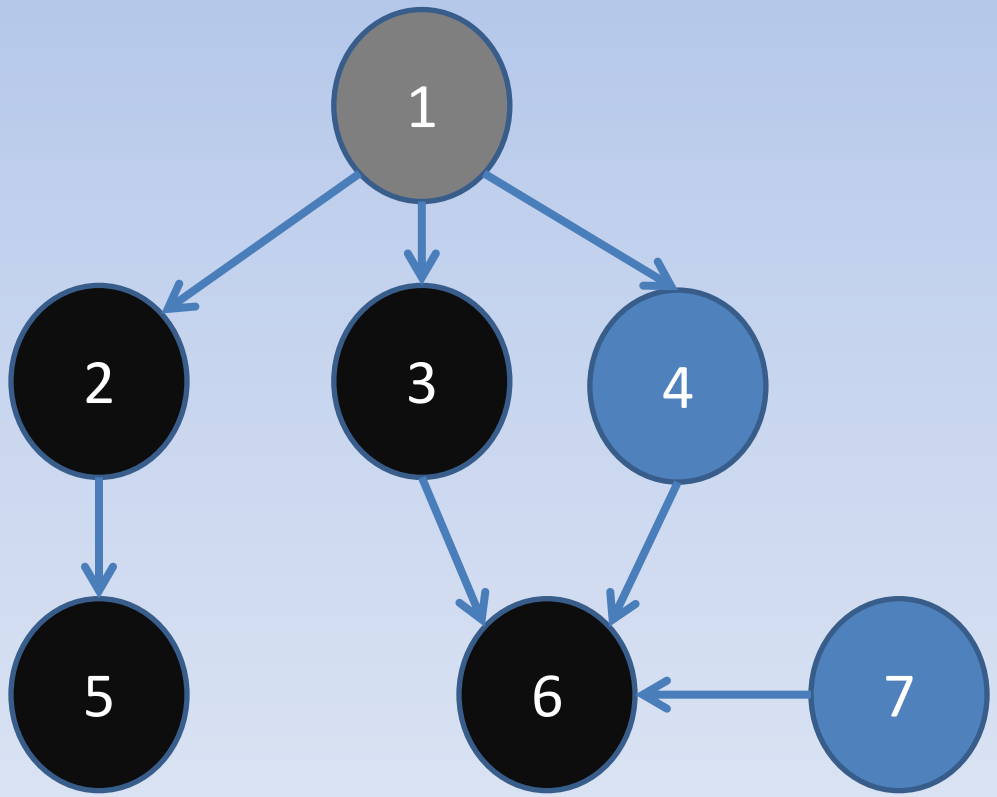
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6		
4			
5	3	4	
6	7	8	
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



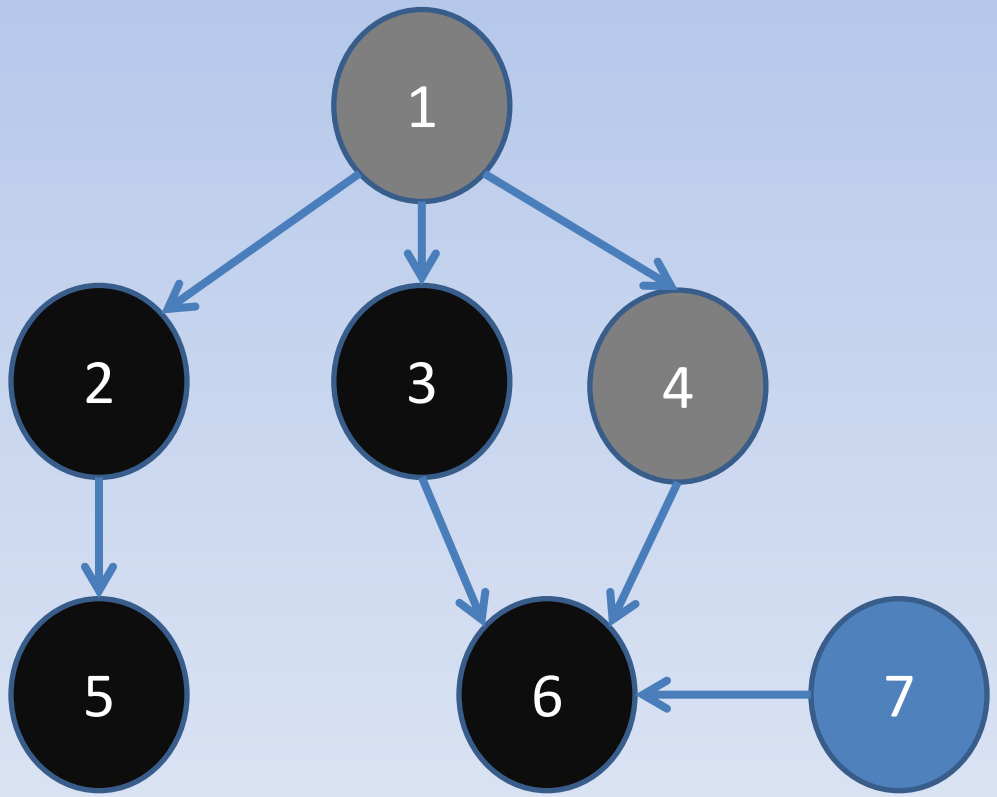
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6	9	
4			
5	3	4	
6	7	8	
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



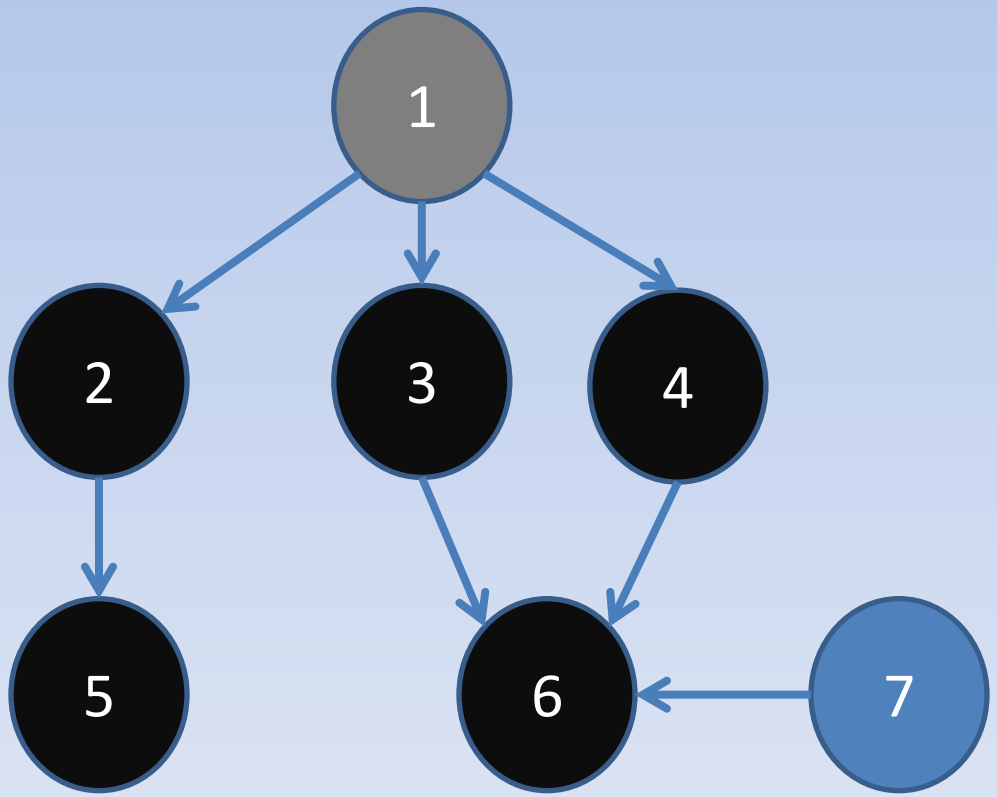
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6	9	
4	10		
5	3	4	
6	7	8	
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



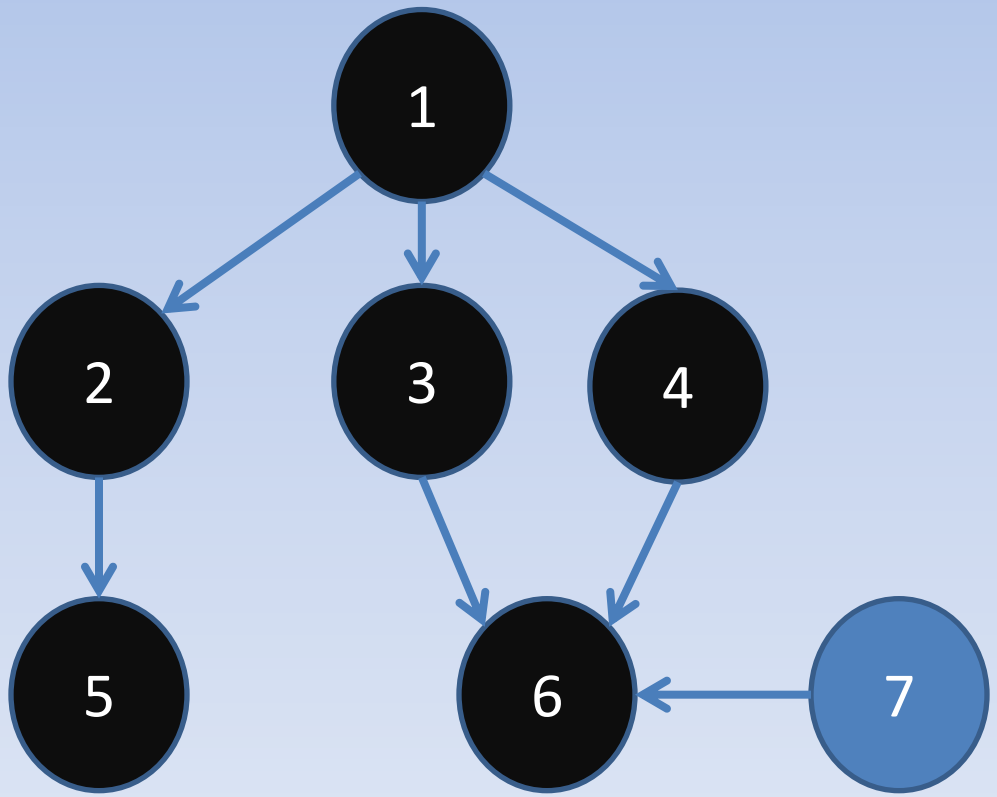
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1		
2	2	5	
3	6	9	
4	10	11	
5	3	4	
6	7	8	
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



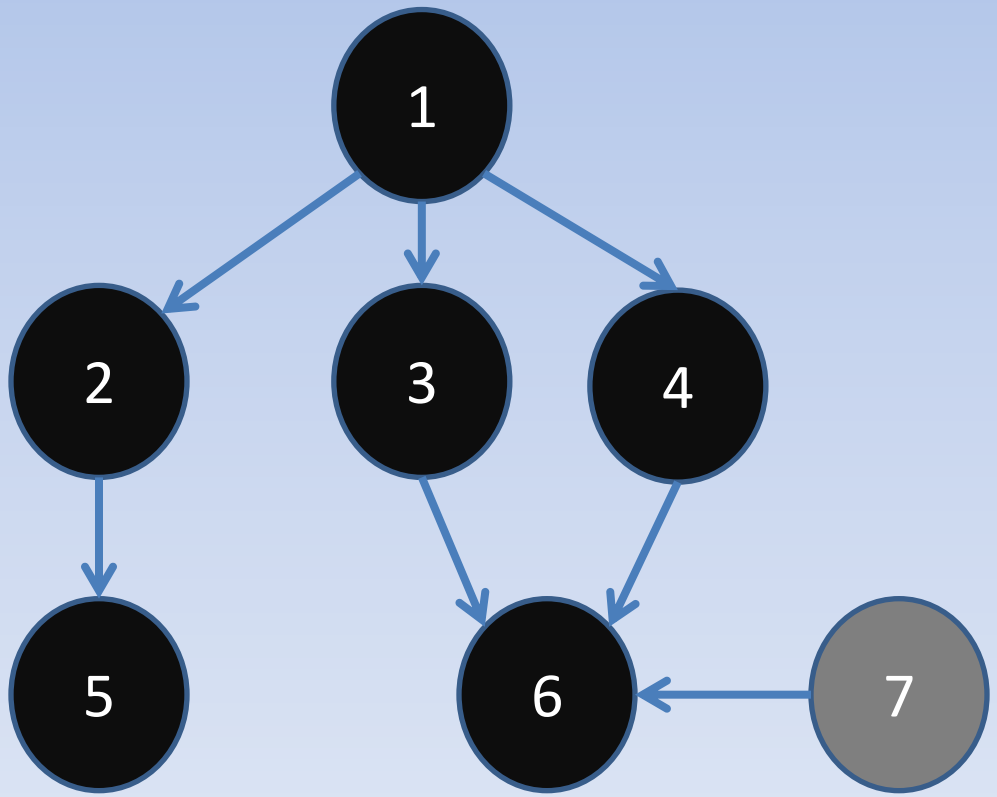
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1	12	
2	2	5	
3	6	9	
4	10	11	
5	3	4	
6	7	8	
7			

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



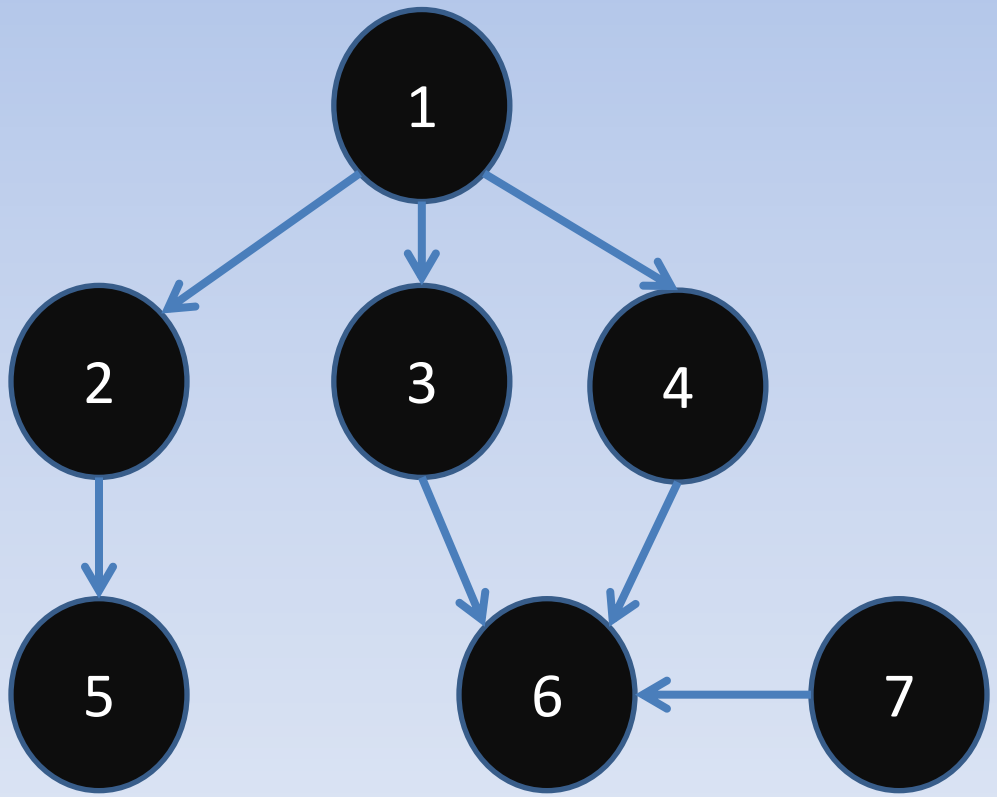
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1	12	
2	2	5	
3	6	9	
4	10	11	
5	3	4	
6	7	8	
7	13		

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1	12	
2	2	5	
3	6	9	
4	10	11	
5	3	4	
6	7	8	
7	13	14	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

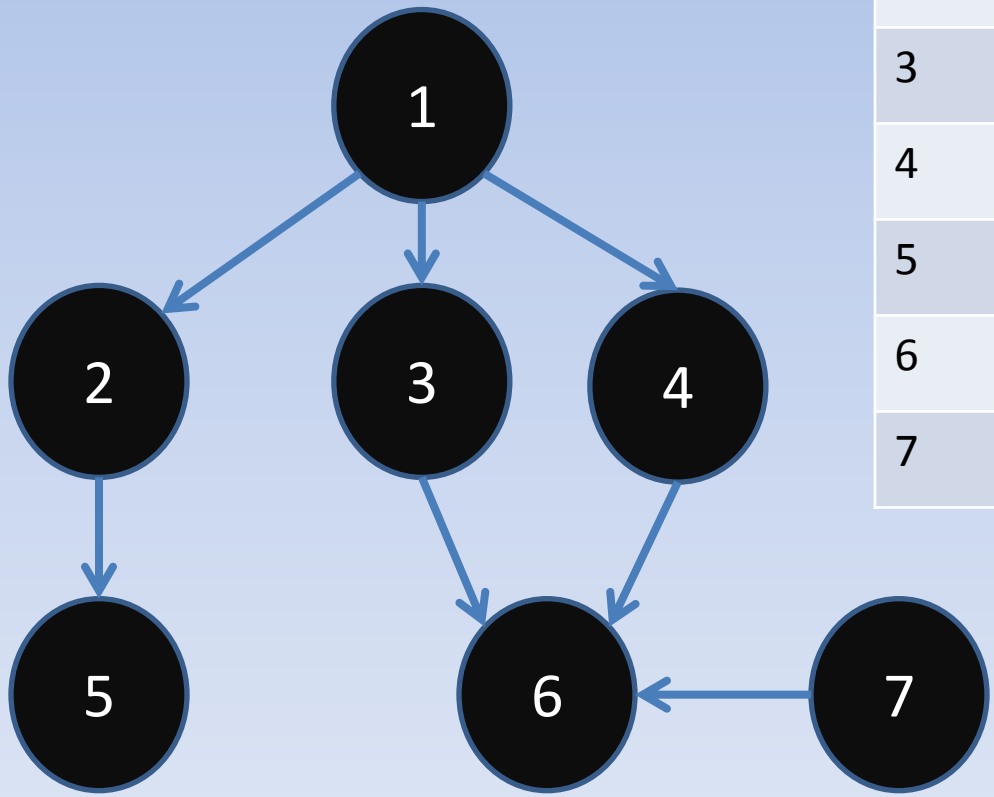
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Discover Time	Finish Time	Topological Order
1	1	12	2
2	2	5	6
3	6	9	4
4	10	11	3
5	3	4	7
6	7	8	5
7	13	14	1

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

Vertex	Topological Order
1	2
2	6
3	4
4	3
5	7
6	5
7	1



Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]

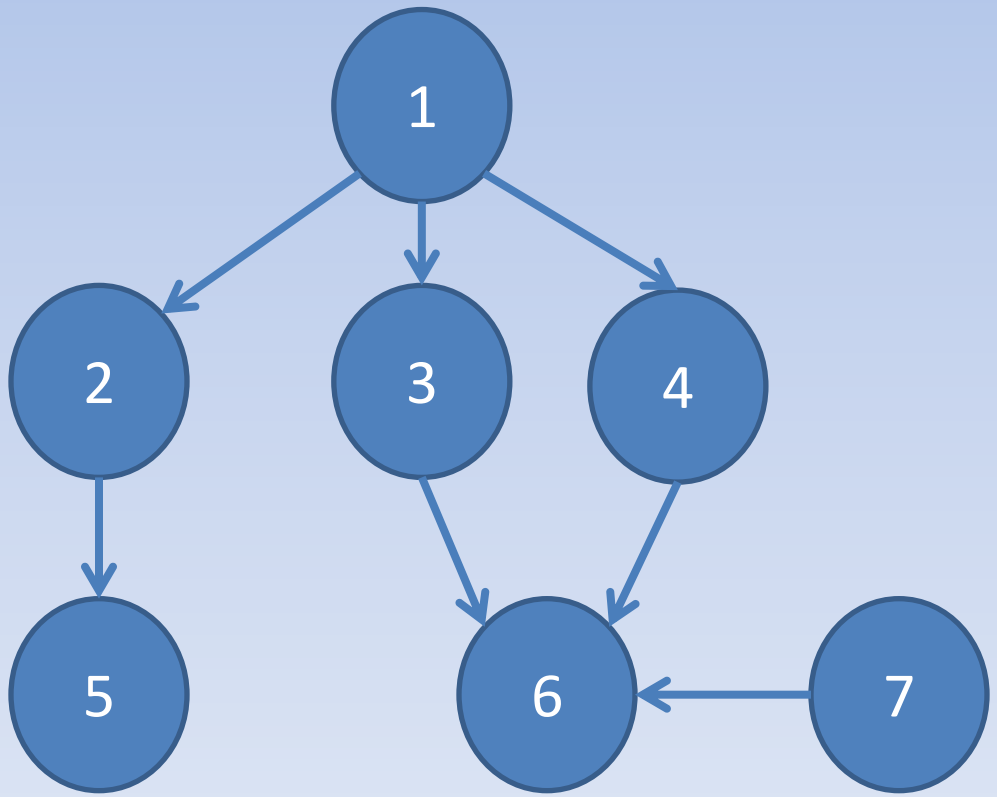
Edge	Edge Type
(1, 2)	
(1, 3)	
(1, 4)	
(2, 5)	
(3, 6)	
(4, 6)	
(7, 6)	

Determining Edge Types

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

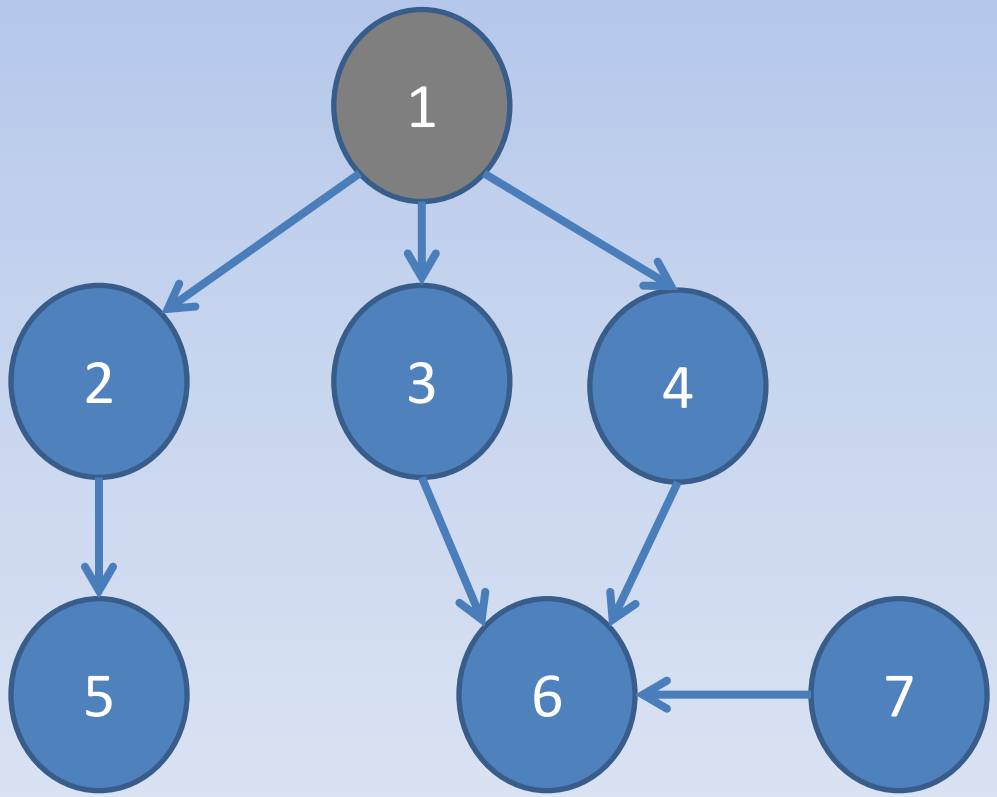
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	
(1, 3)	
(1, 4)	
(2, 5)	
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

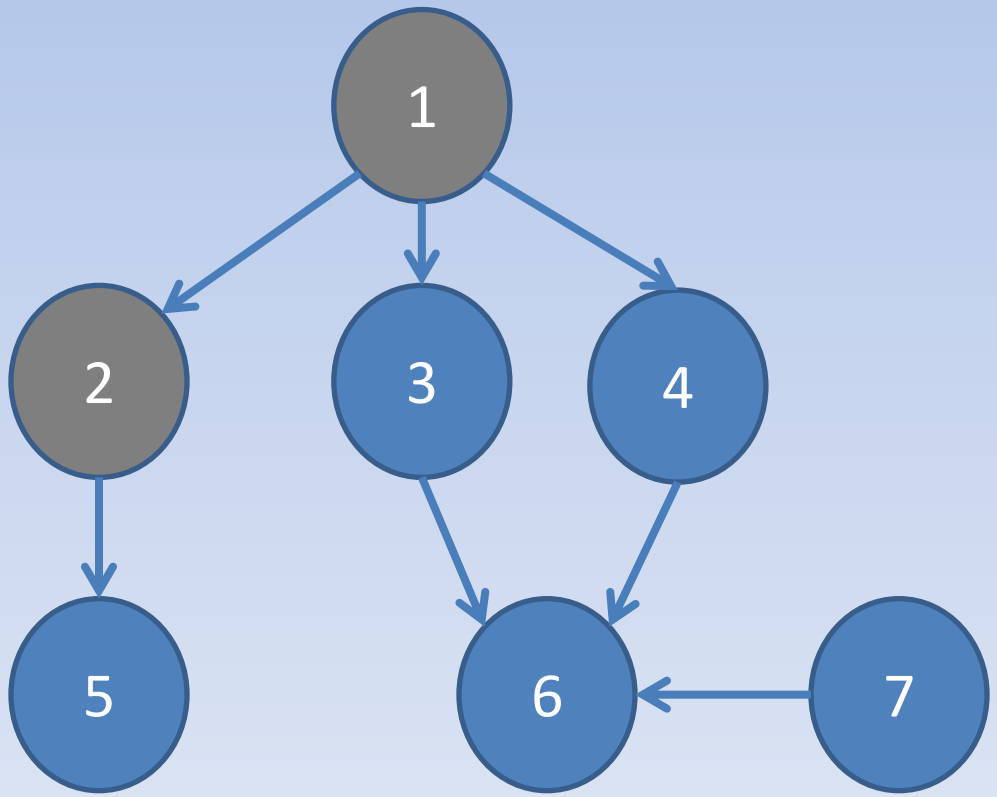
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	
(1, 3)	
(1, 4)	
(2, 5)	
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

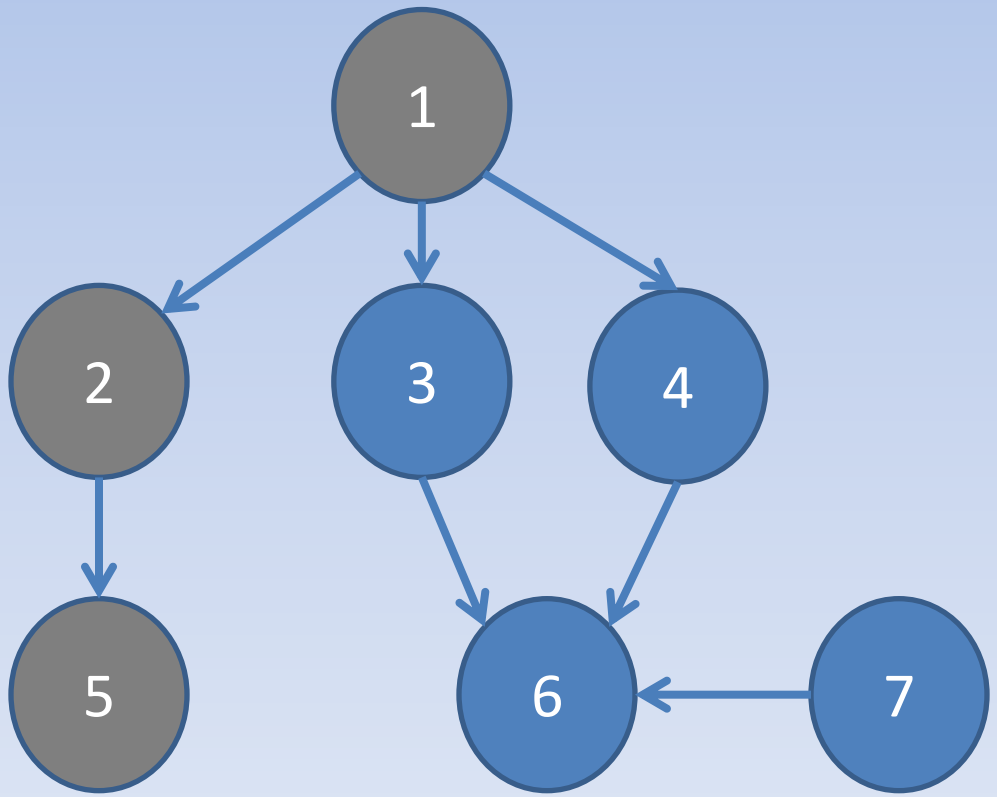
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	
(1, 4)	
(2, 5)	
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

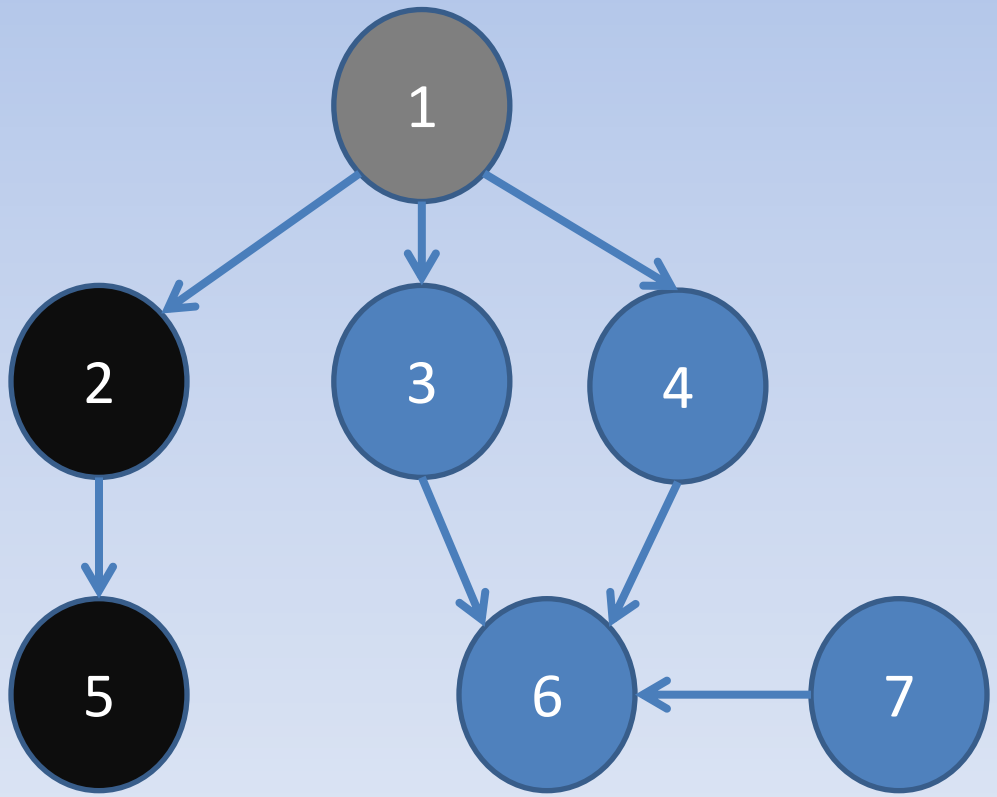
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	
(1, 4)	
(2, 5)	Tree
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

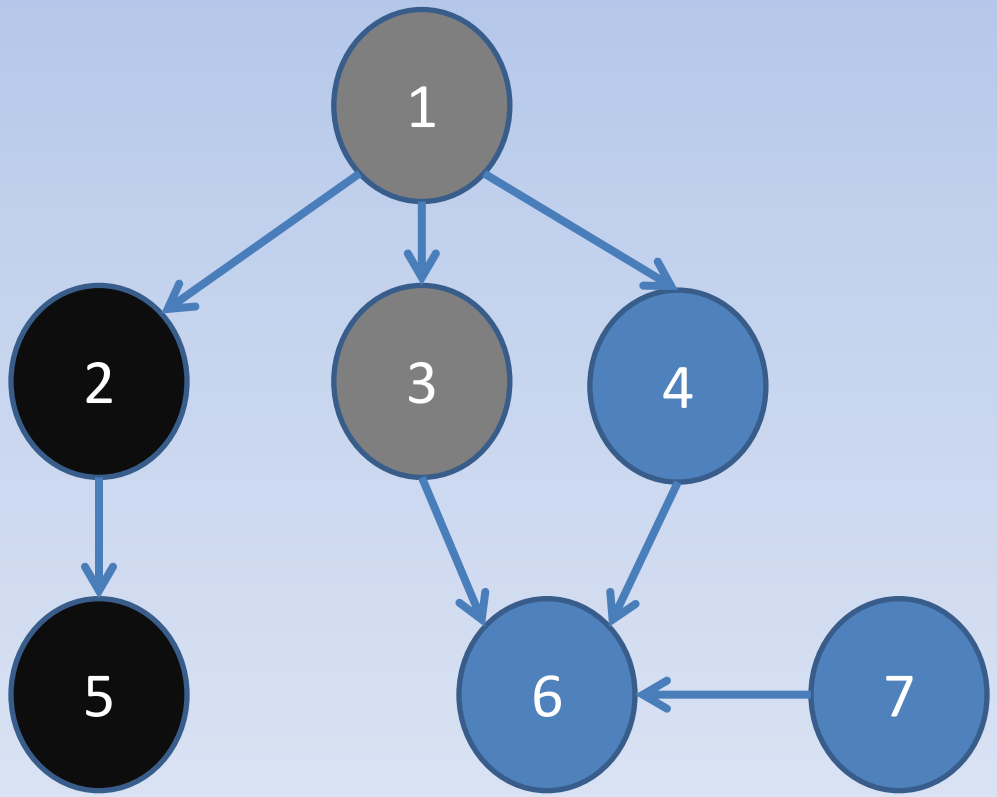
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	
(1, 4)	
(2, 5)	Tree
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

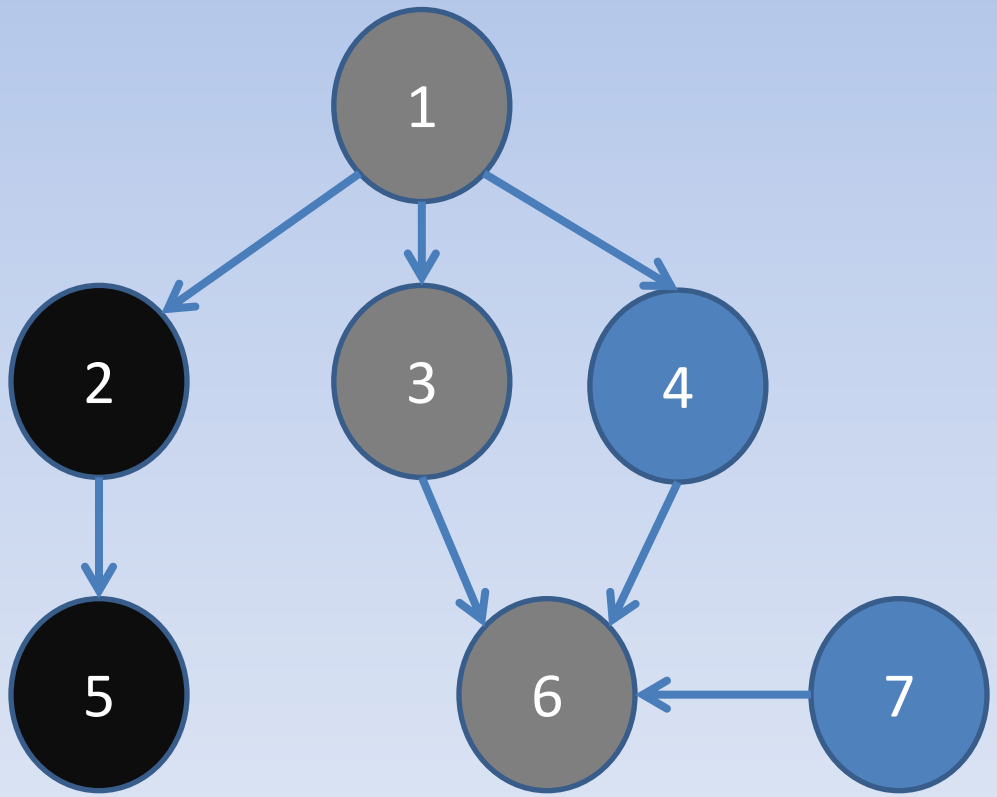
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	
(2, 5)	Tree
(3, 6)	
(4, 6)	
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

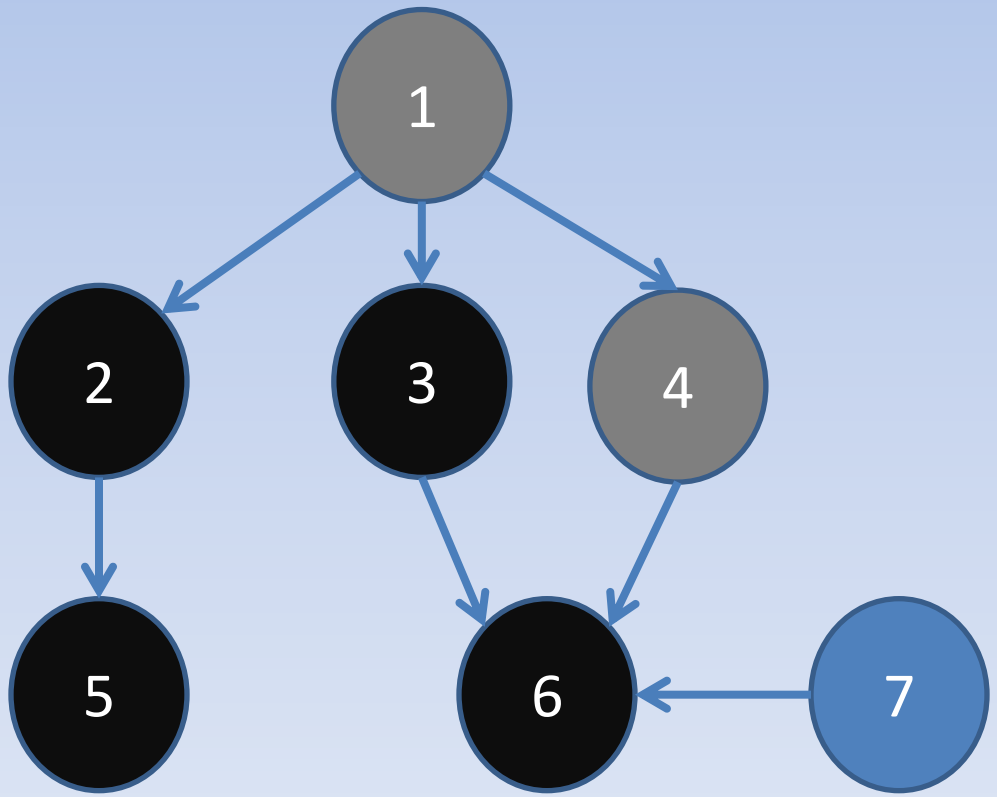
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	
(7, 6)	

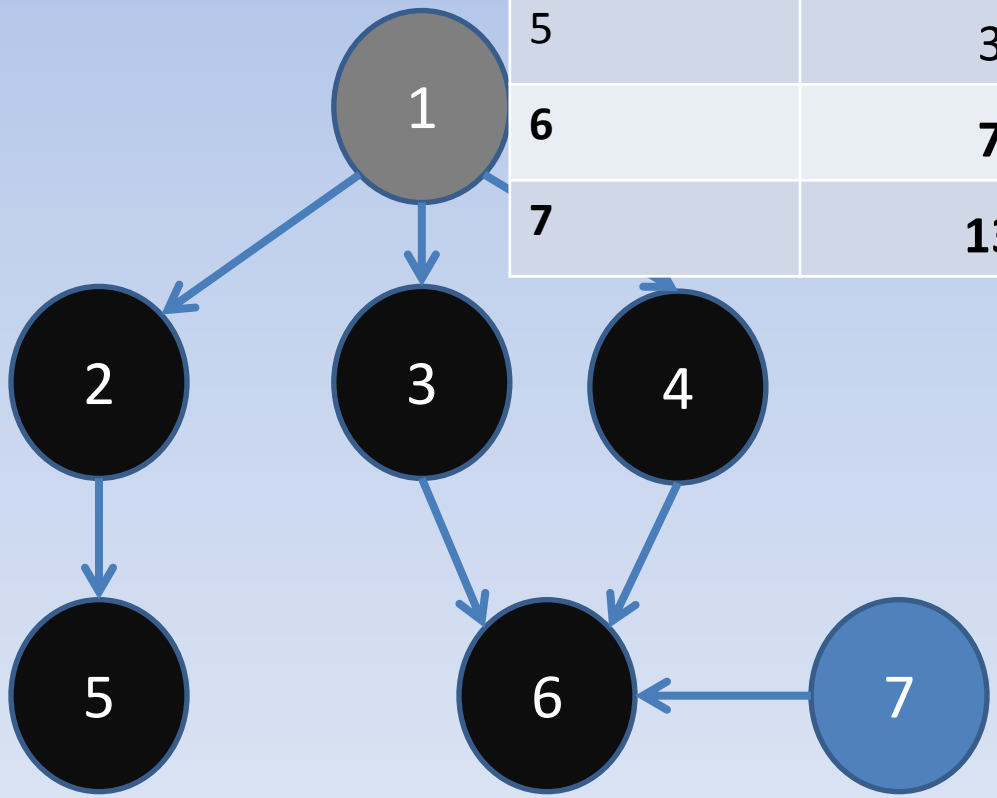
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	
(7, 6)	

Vertex	Discover Time
1	1
2	2
3	6
4	10
5	3
6	**7**
7	**13**

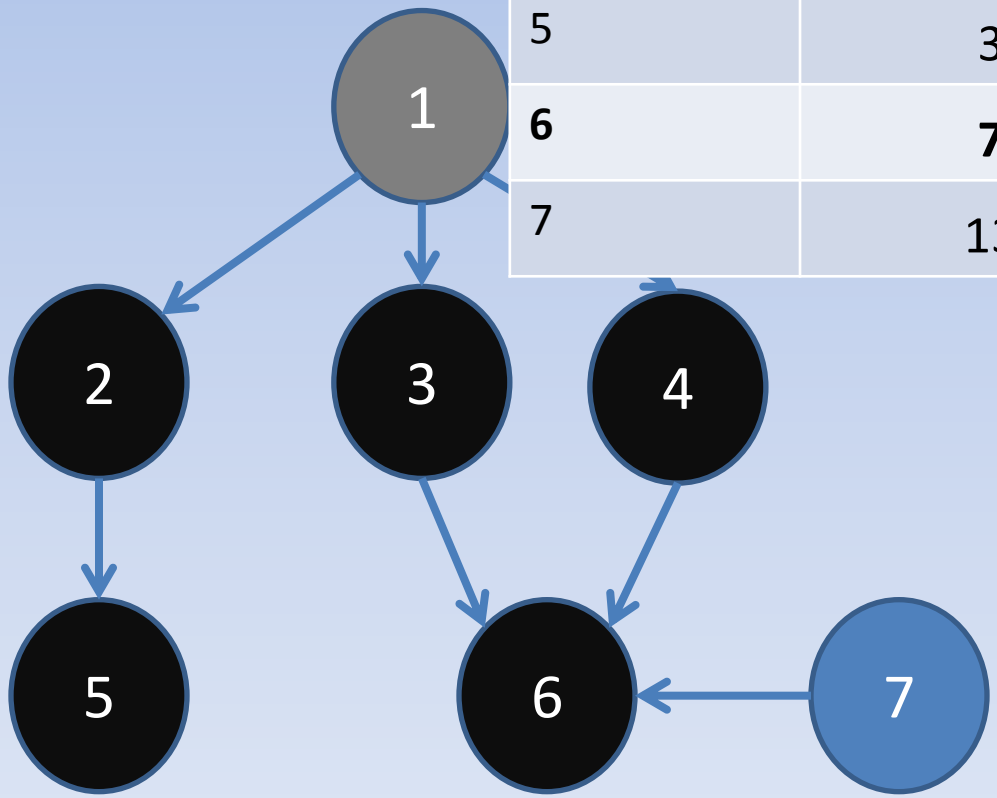


Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	
(7, 6)	

Determining Edge Types

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

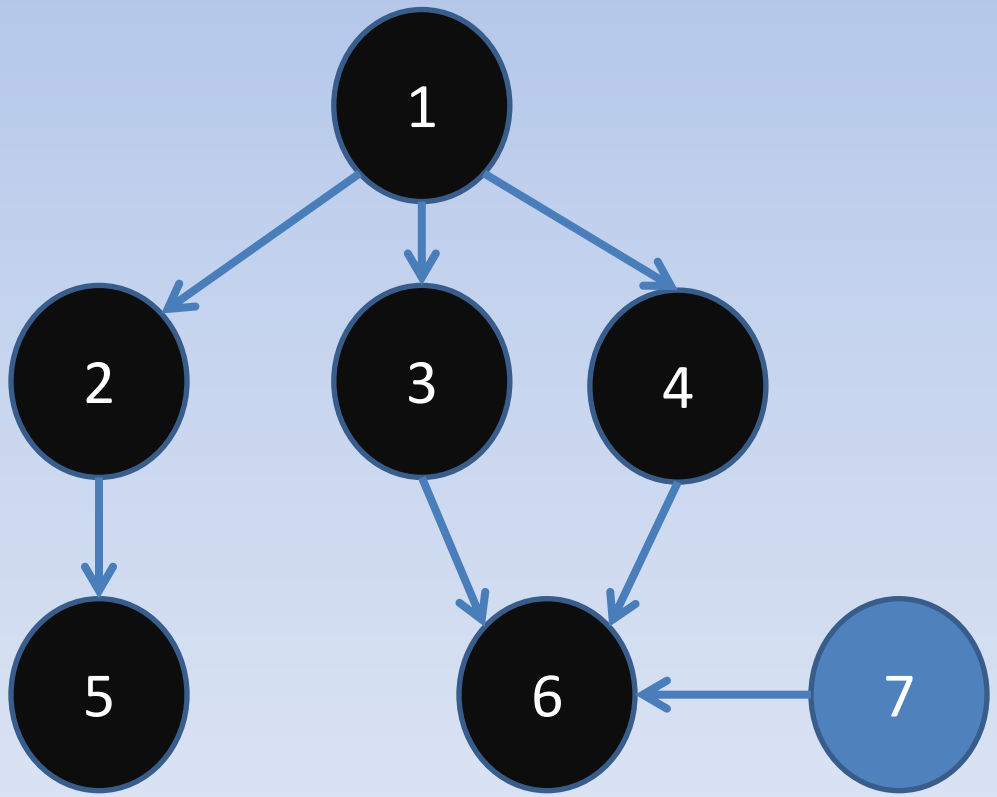
Vertex	Discover Time
1	1
2	2
3	6
4	**10**
5	3
6	**7**
7	13



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	Cross
(7, 6)	

Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

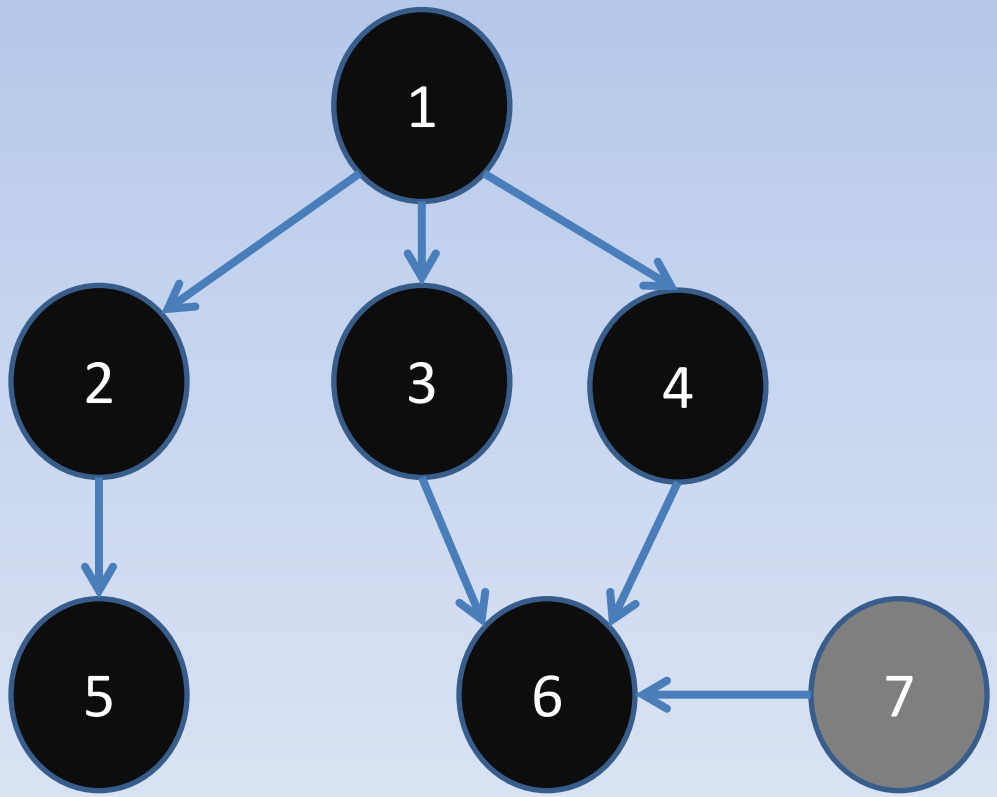
1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	Cross
(7, 6)	

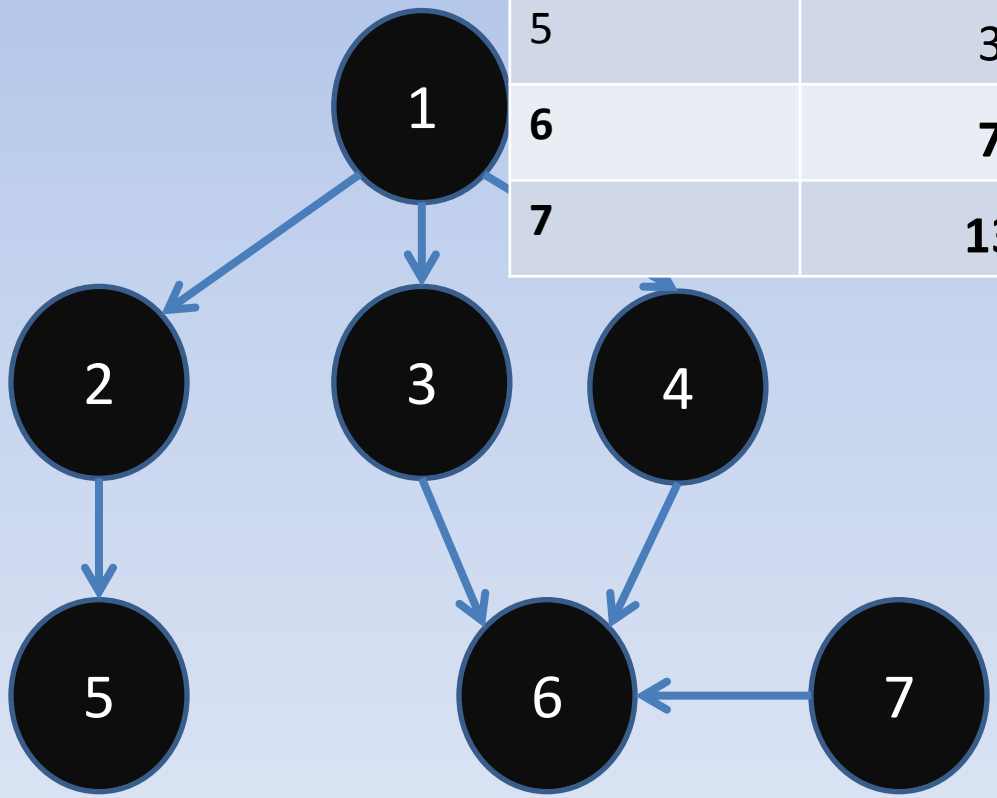
Given Graph G with V={1,2,3,4,5,6,7} and Adjacency List below:

1 ==> [2, 3, 4]
2 ==> [5]
3 ==> [6]
4 ==> [6]
7 ==> [6]



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	Cross
(7, 6)	

Vertex	Discover Time
1	1
2	2
3	6
4	10
5	3
6	**7**
7	**13**

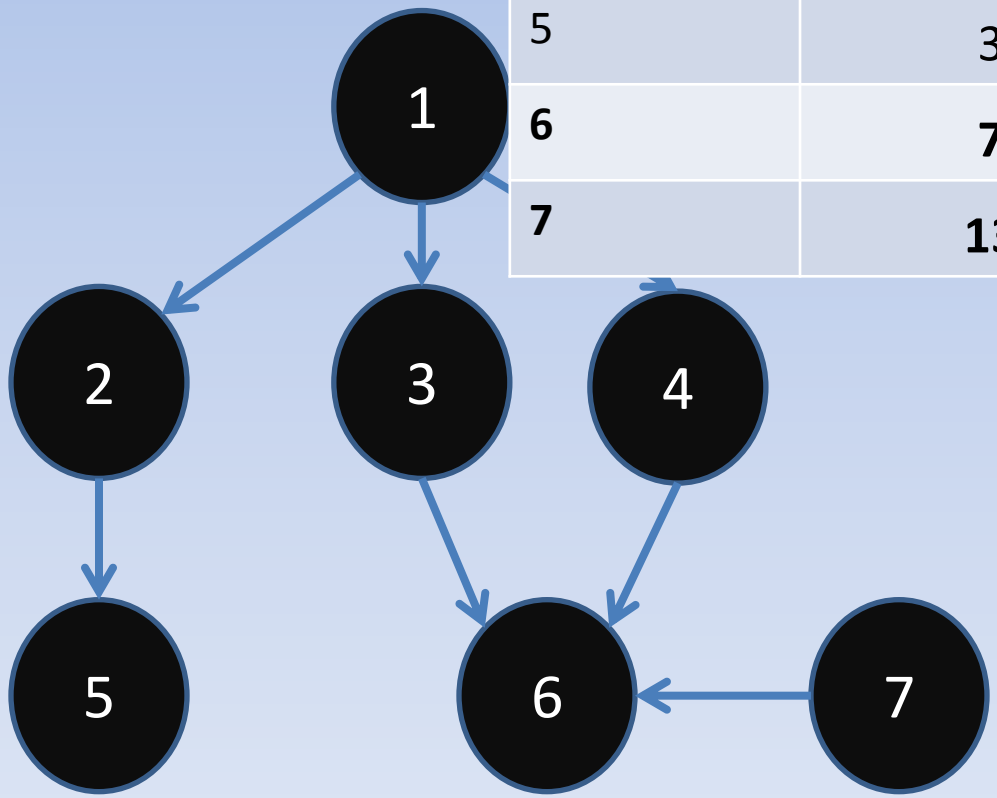


Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	Cross
(7, 6)	

Determining Edge Types

- When first exploring an edge (u, v)
- If color V:
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - u.d < v.d: forward edge
    - u.d > v.d: cross edge

Vertex	Discover Time
1	1
2	2
3	6
4	10
5	3
6	**7**
7	**13**



Edge	Edge Type
(1, 2)	Tree
(1, 3)	Tree
(1, 4)	Tree
(2, 5)	Tree
(3, 6)	Tree
(4, 6)	Cross
(7, 6)	Cross

Design and Analysis of Algorithms

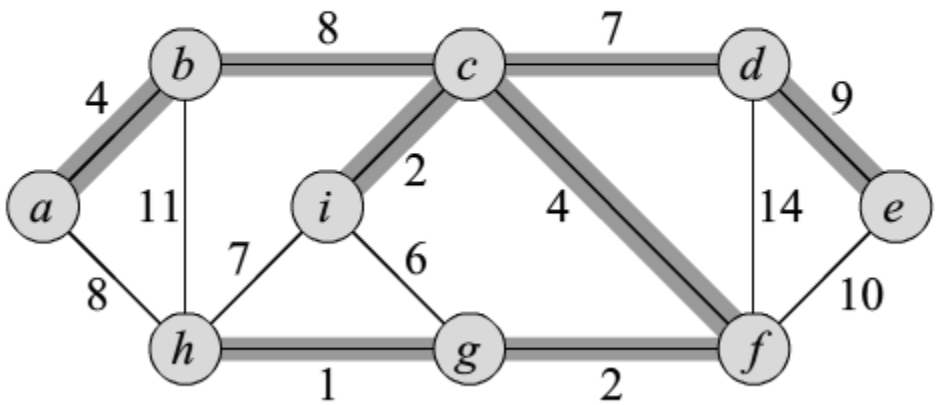
Section VI : Graph Algorithms

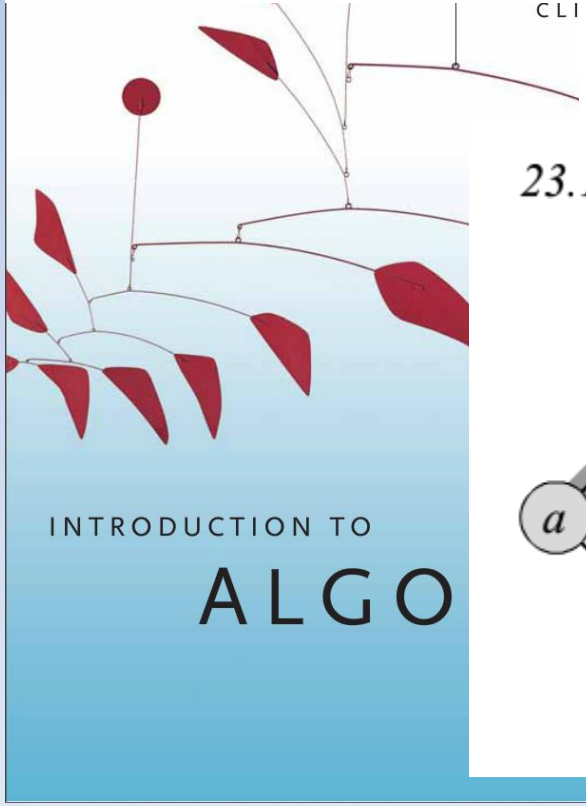
Chapter 23: Minimum Spanning Trees

VI Graph Algorithms

23 Minimum Spanning Trees

23.1 Growing a minimum spanning tree





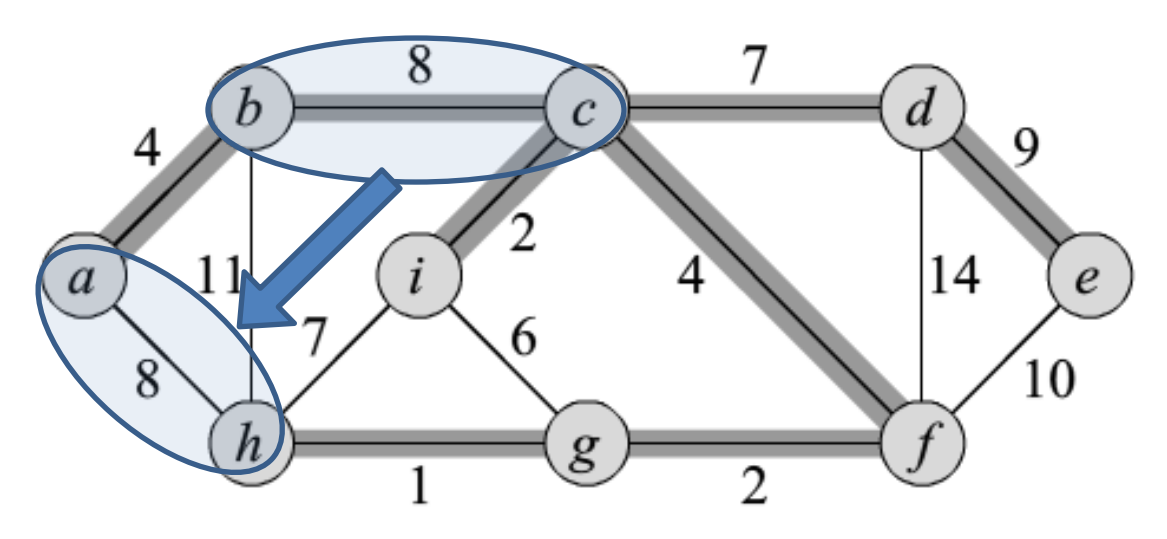
Minimum Spanning Tree

- Given Graph G=(V, E)
  - V is set of pints
  - E is possible interconnections
- Weight Function w(u,v) specifying cost to connect u & v.
- We wish to find an acyclic subset $T \subseteq E$ connecting all vertices and minimizing total weight:

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

- Since T is acyclic and connects all vertices, it must form a tree.
- Since it spans the graph, we call it a Spanning Tree.

MST



- MST cost of 37
- (b, c) => (a, h)
  - Another Minimum-Spanning Tree

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

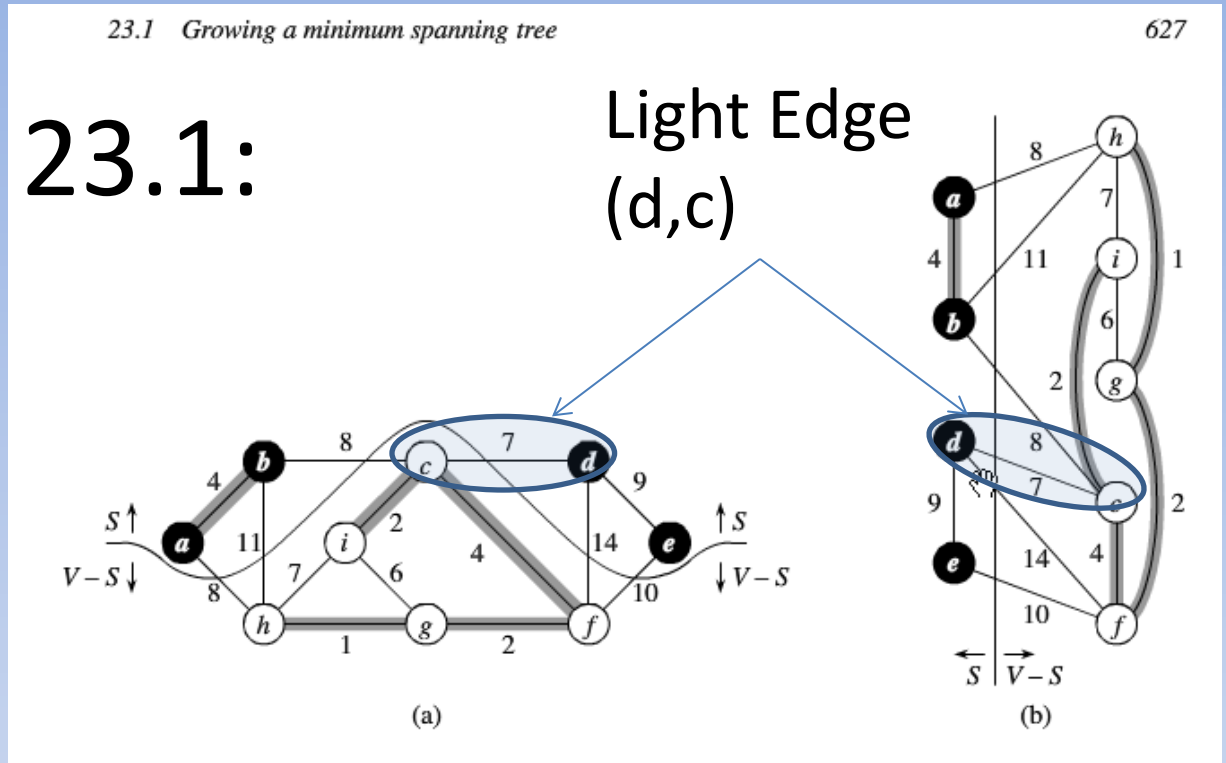
Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

- Problem is finding SAFE EDGE!

Finding Safe Edges
First: Some Definitions (Of Course!)

- A **cut** (S, V – S) of an undirected Graph G=(V,E) is a partition of V.
- Edge (u,v) ∈ E **crosses** the cut if one if its endpoints is in S and the other is in (V-S).
- A cut **respects** a set A of edges if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is minimum of any edge crossing the cut!

Theorem 23.1:



- Let G=(V,E) be a connected, undirected graph with real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G
- Let (S, V-S) be any cut of G that respects A
- Let (u,v) be a light edge crossing (S, V-S).
- THEN: (u,v) is safe for A.

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- Depends upon Union-Find
  - Chapter 21

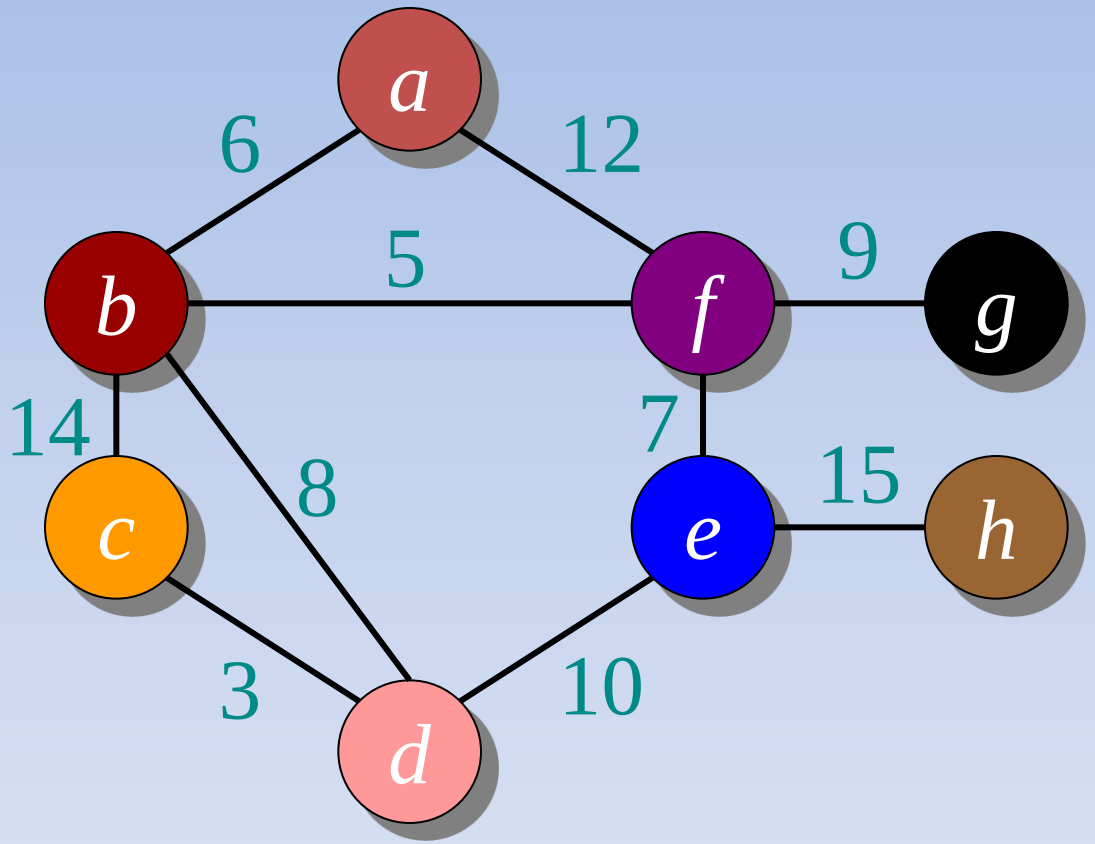
21 Data Structures for Disjoint Sets

Kruskal's algorithm in words

- Procedure:
  - Sort all edges into non-decreasing order
  - Initially each node is in its own tree
  - For each edge in the sorted list
    - If the edge connects two separate trees, then
      - join the two trees together with that edge

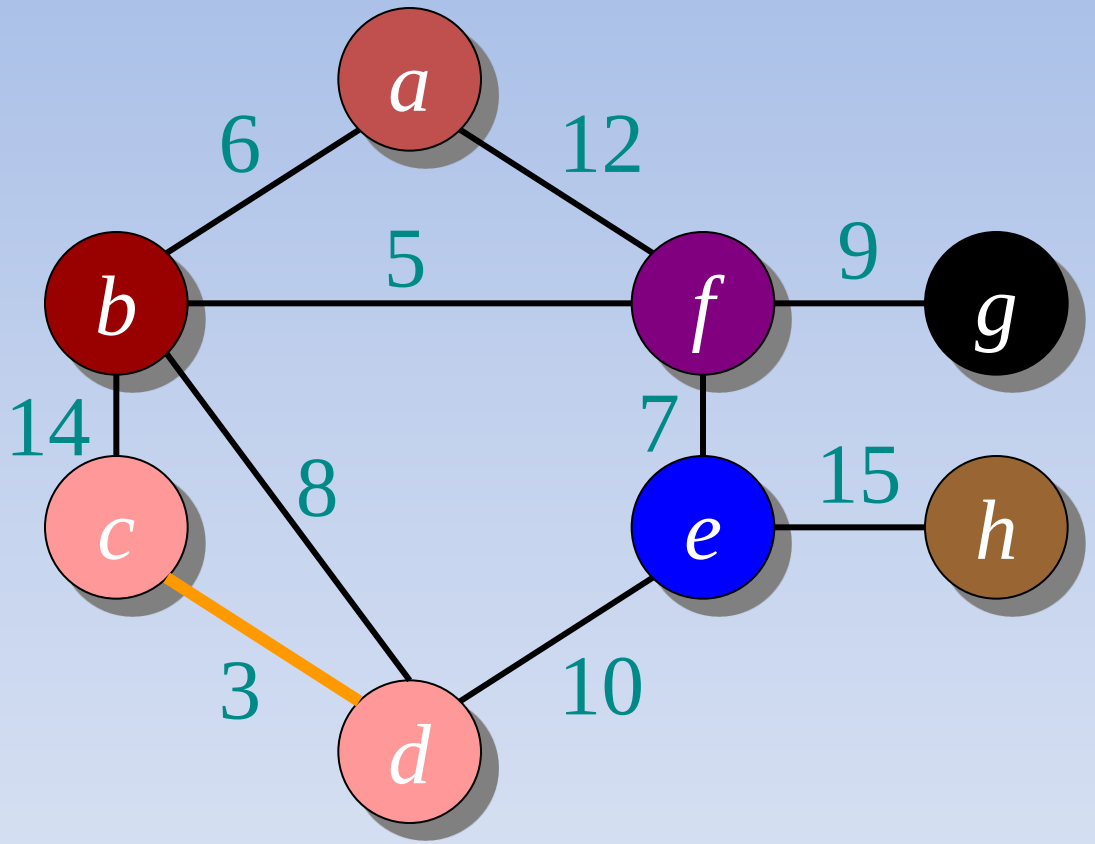
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



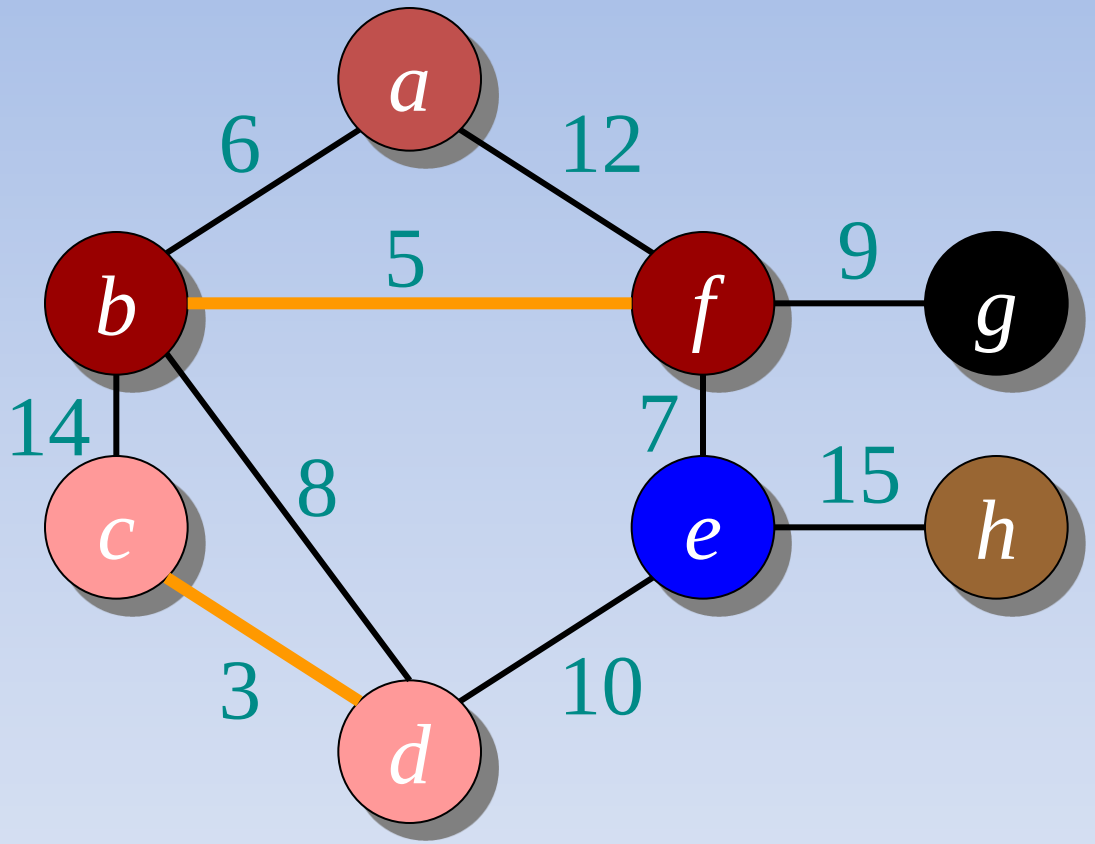
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



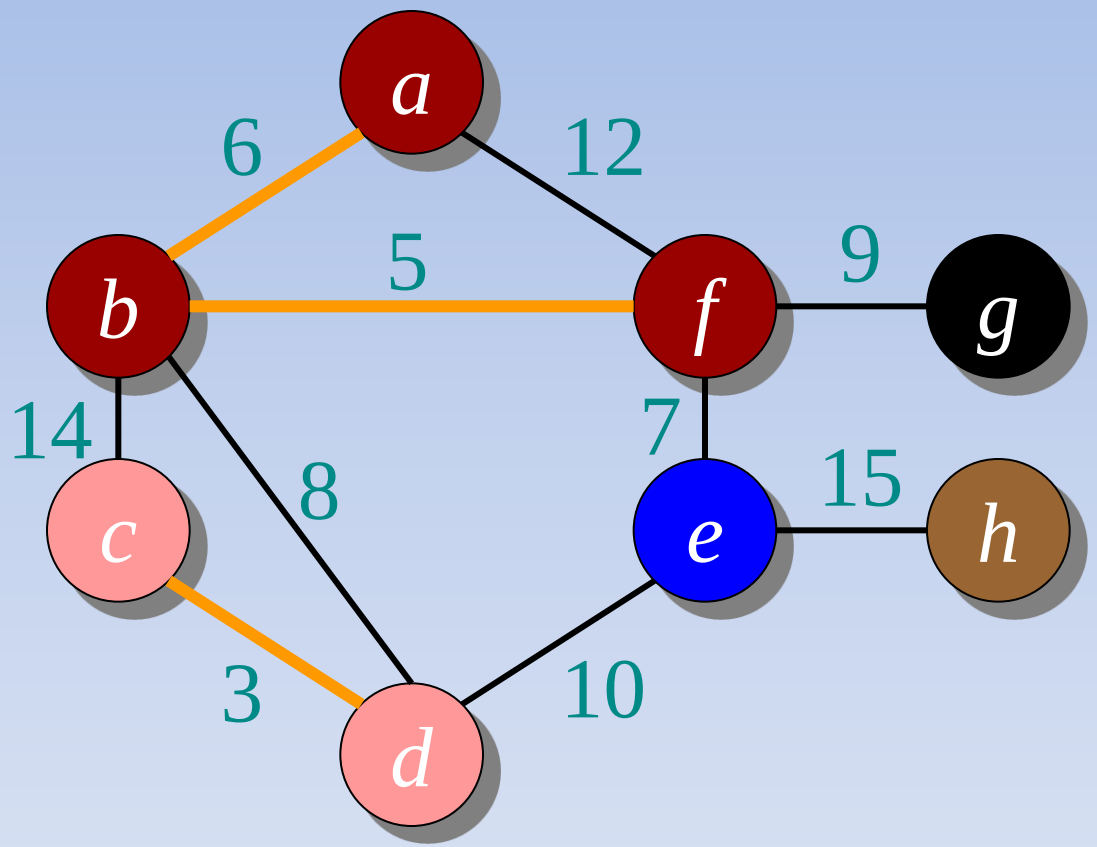
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



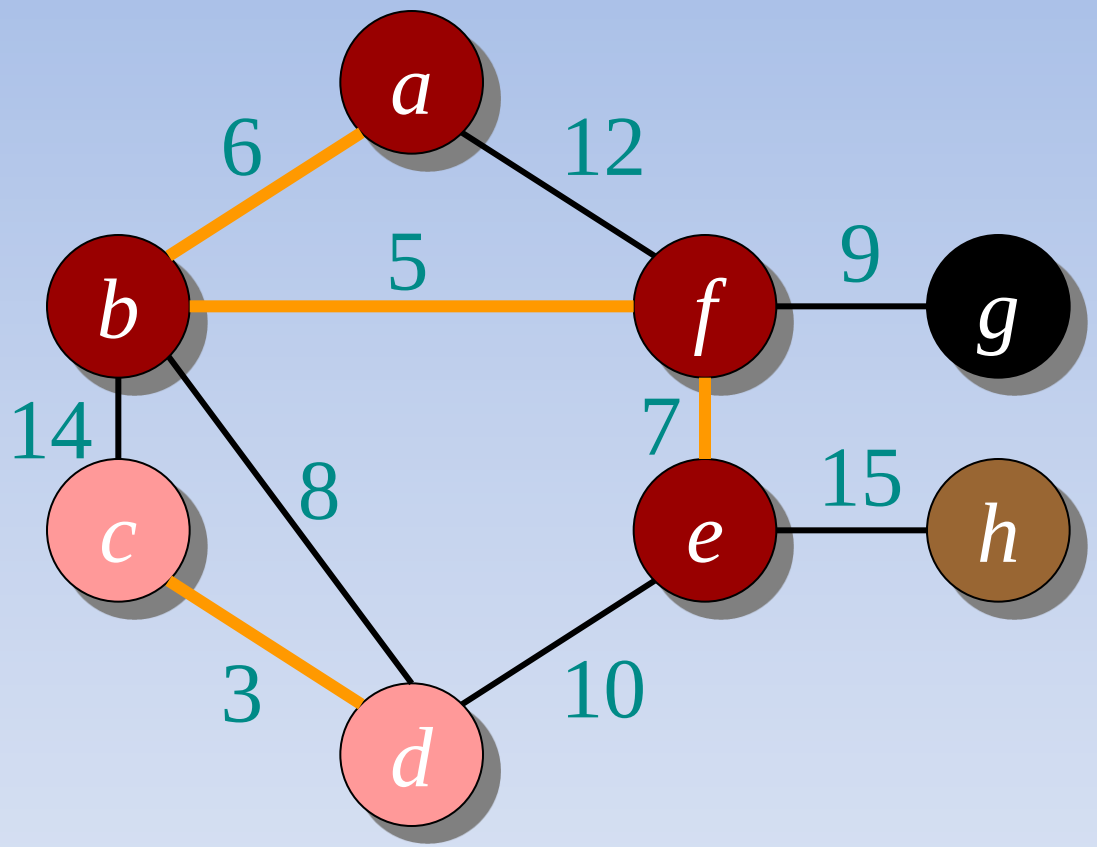
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



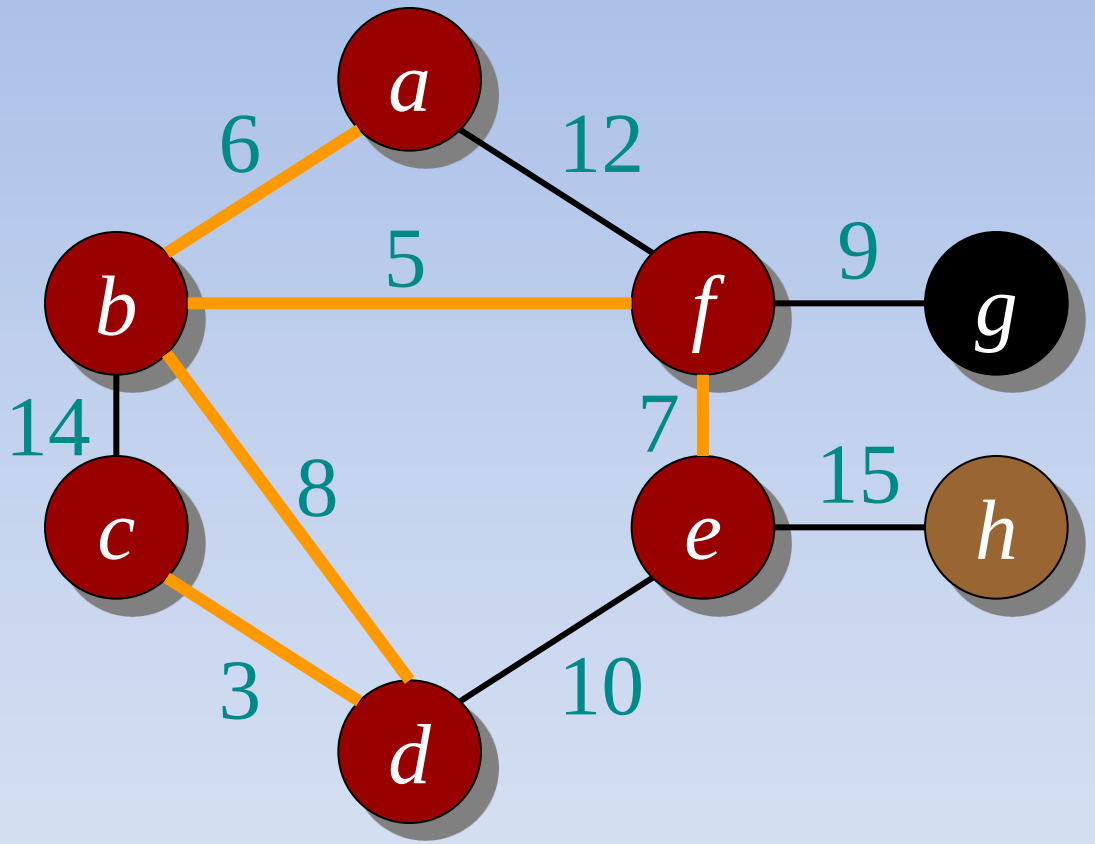
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



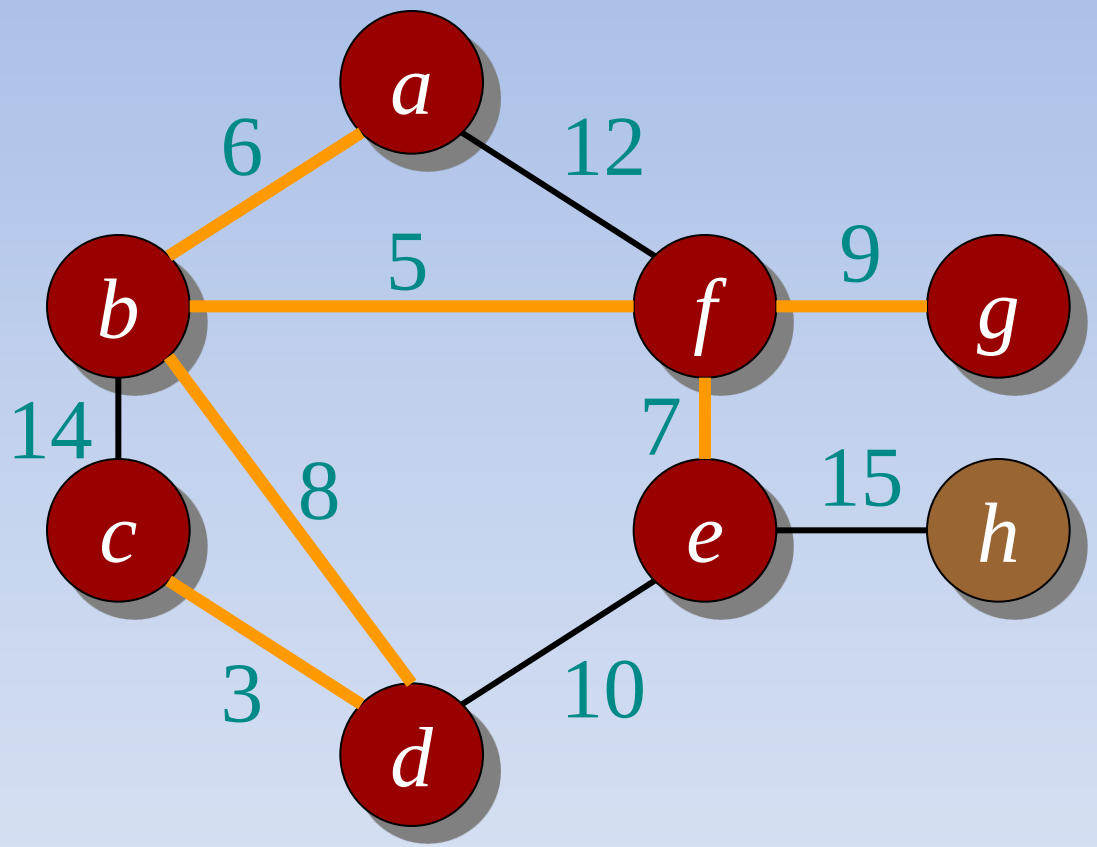
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



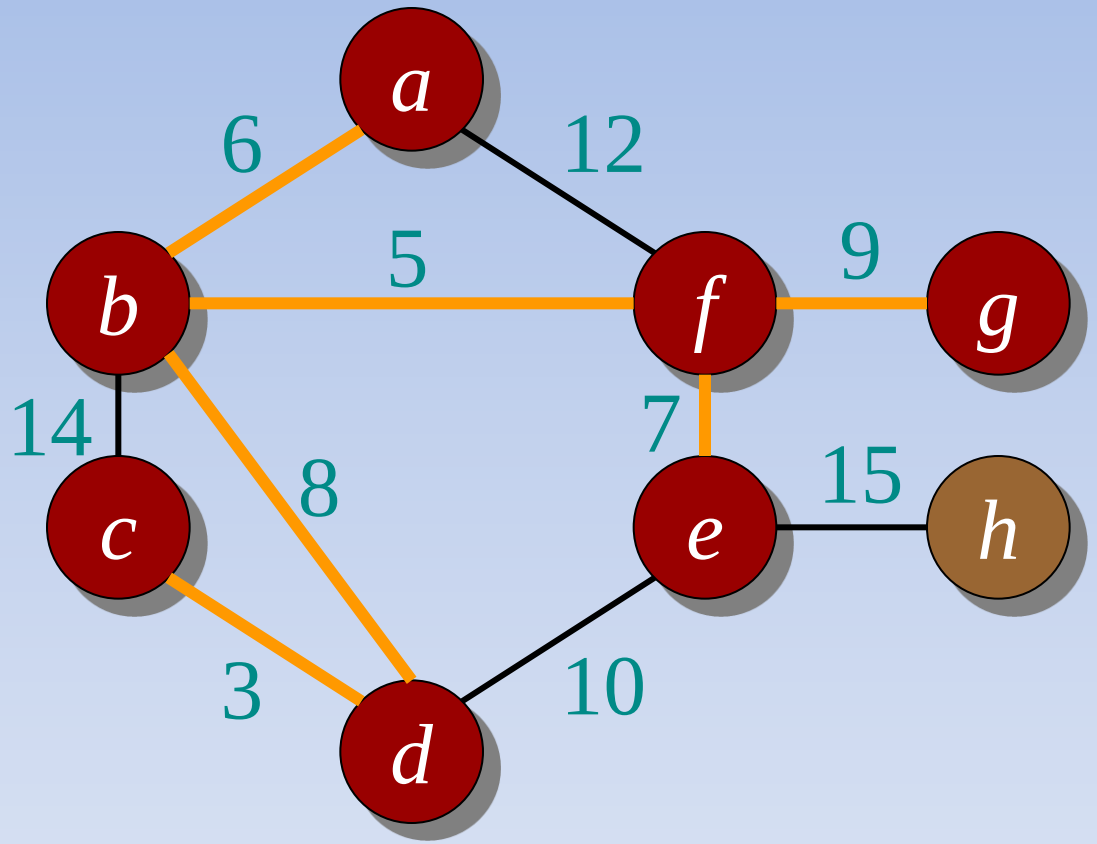
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



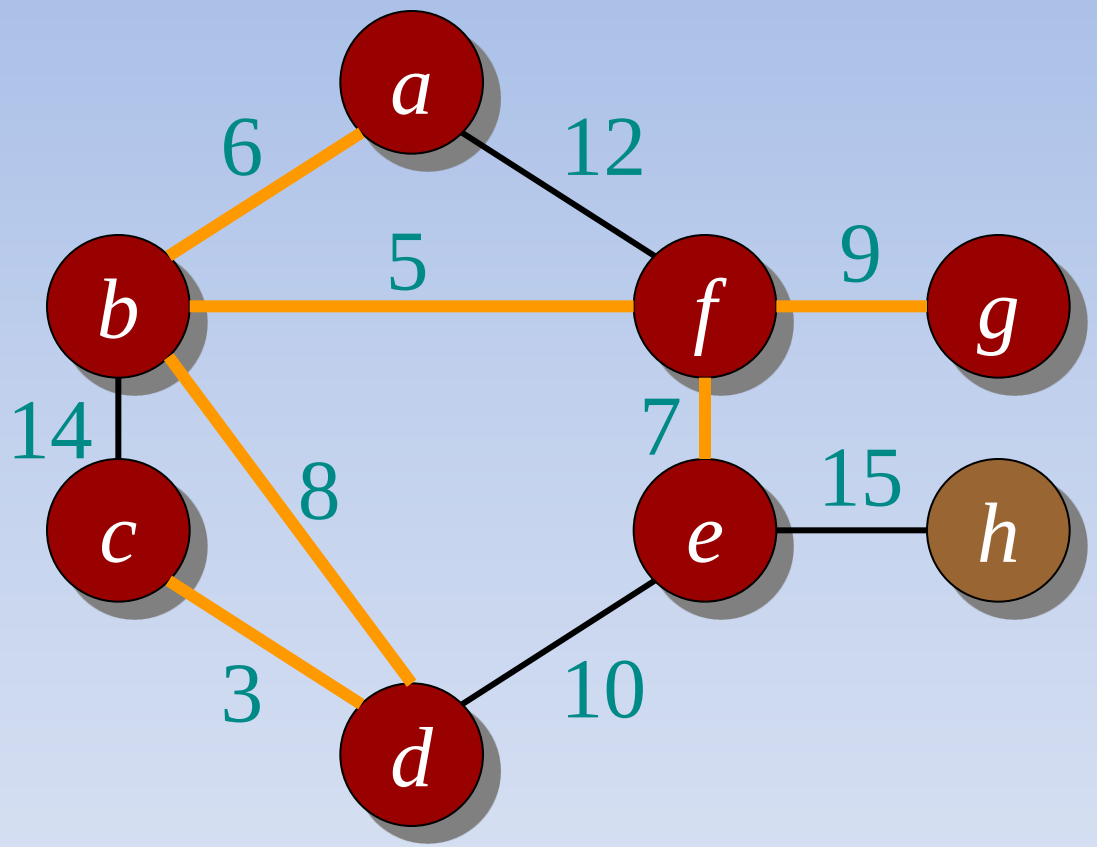
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



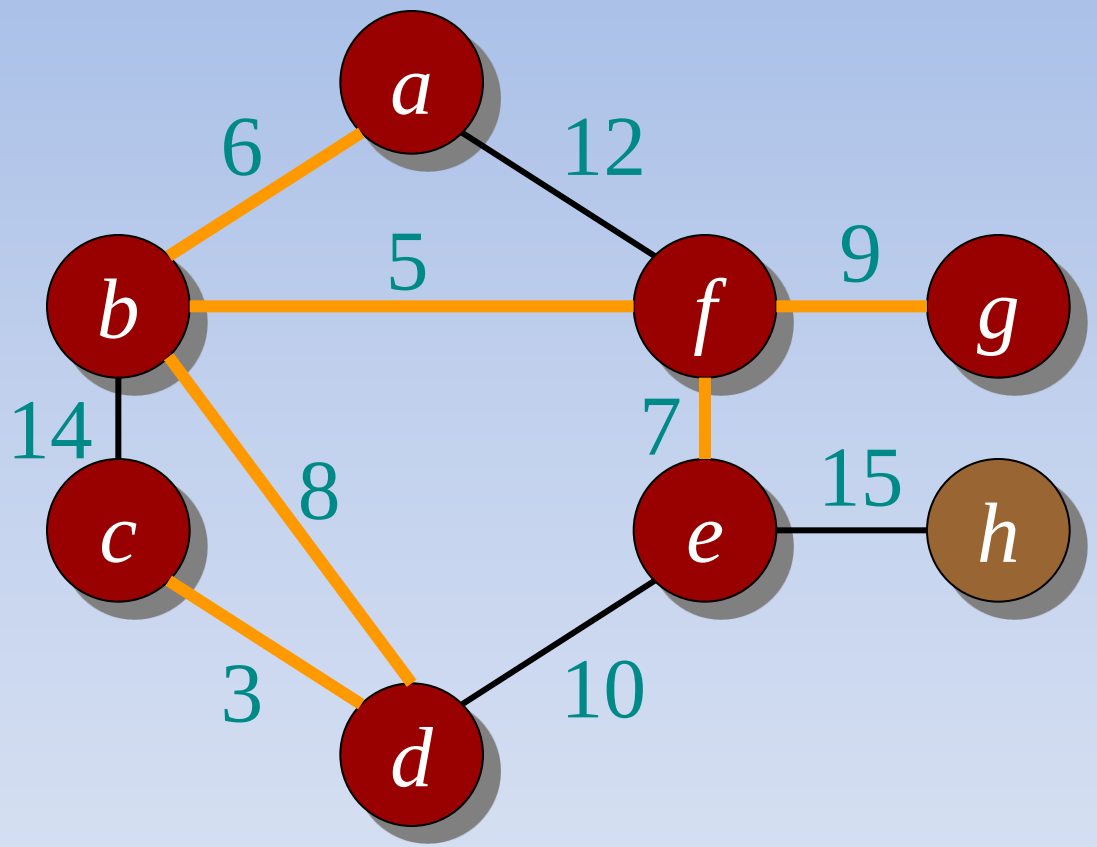
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



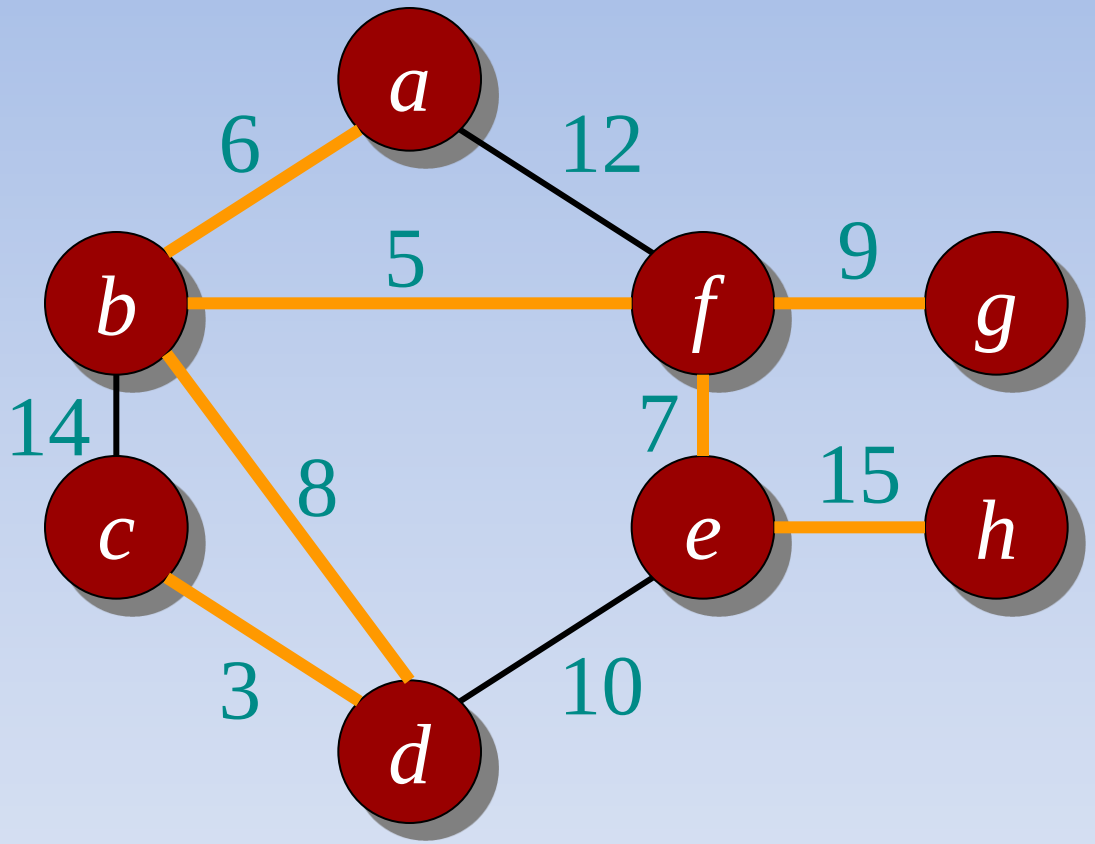
Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Prim's Algorithm

- Greedy Algorithm

- Operates like Dijkstra's Shortest Path (later)
- Algorithm operates so edges in partial solution set of edges always form a single tree.

Prim's Algorithm

- Starts from an arbitrary start node **r**
- Each step adds a LIGHT EDGE to current partial solution set A that connects a to an isolated vertex.

Prim Algorithm

- Q min-priority queue maintains list of all vertices not in tree.
- The MST edge set A is implicitly maintained
- A = { (v, v.p) : v in V – {r} – Q}
- At termination, Q is empty.
- Final MST A = { (v, v.p) for v in V – {r} }

Pseudocode

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Pseudocode

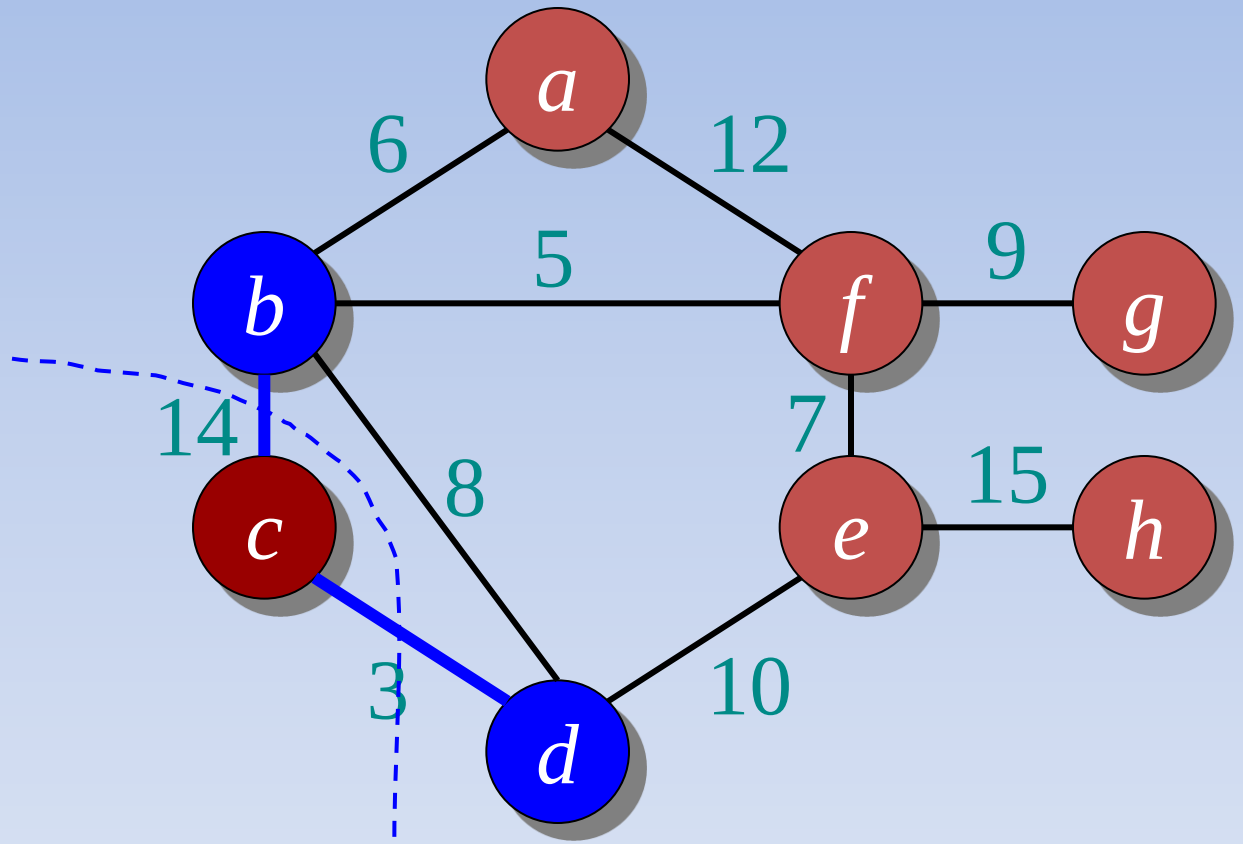
```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Update Key

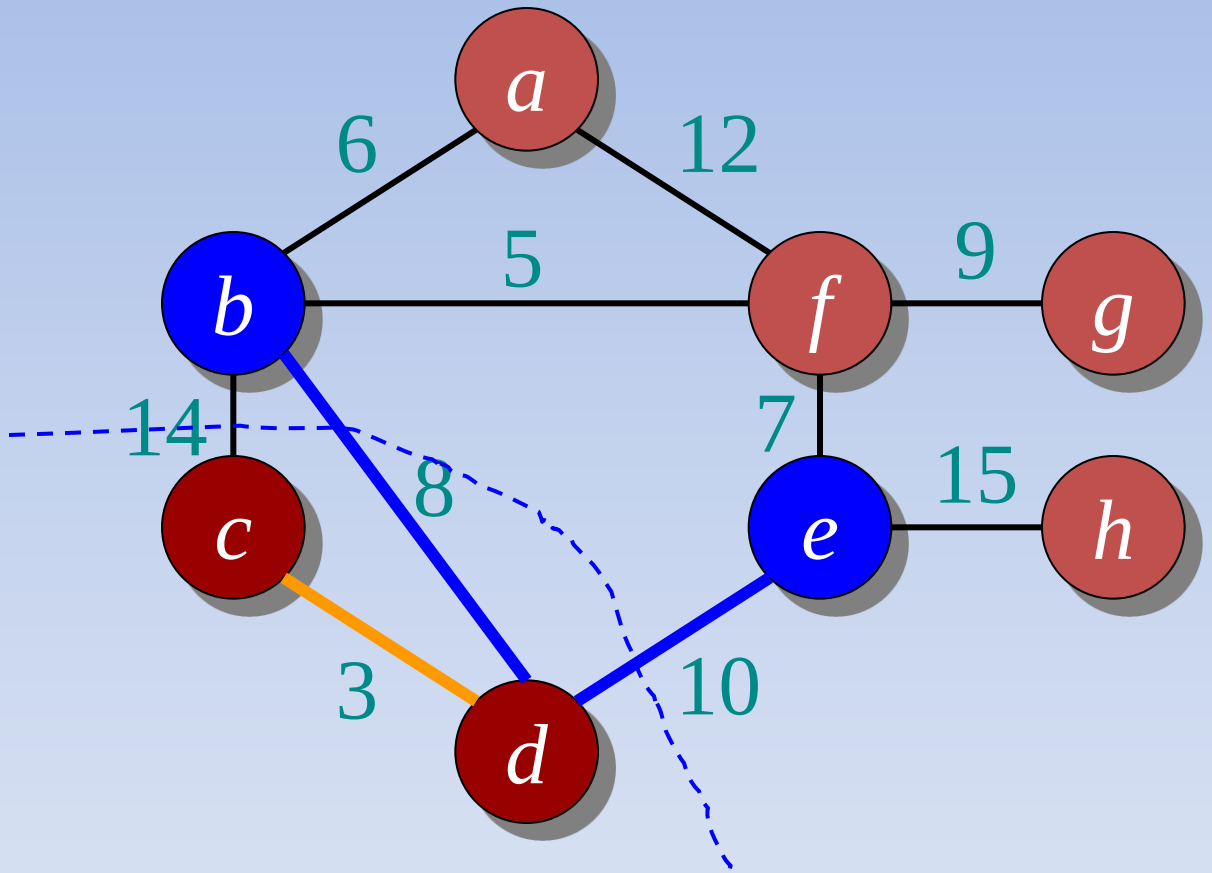
Prim's algorithm in words

- Randomly pick a vertex as the initial tree T
- Gradually expand into a MST:
  - For each vertex that is not in T but directly connected to some nodes in T
    - Compute its minimum distance to any vertex in T
  - Select the vertex that is closest to T
    - Add it to T

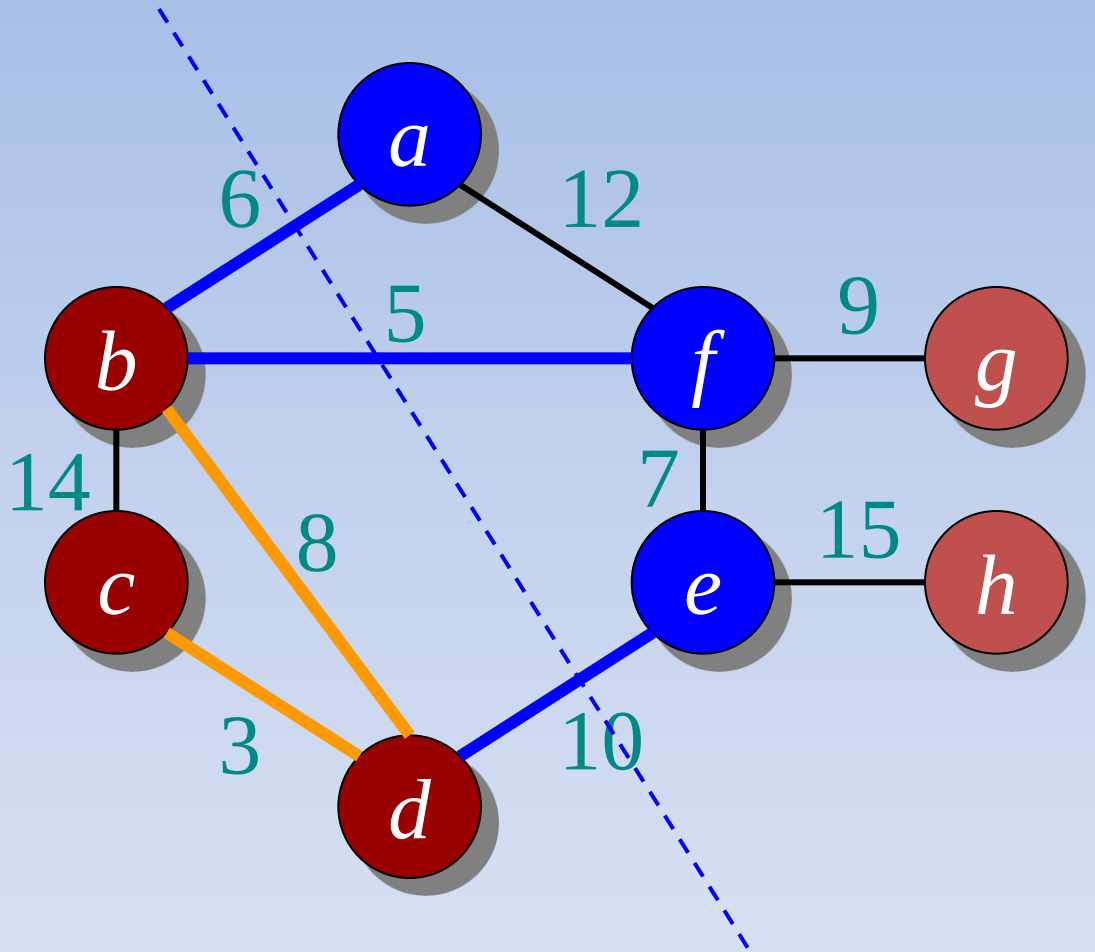
Example



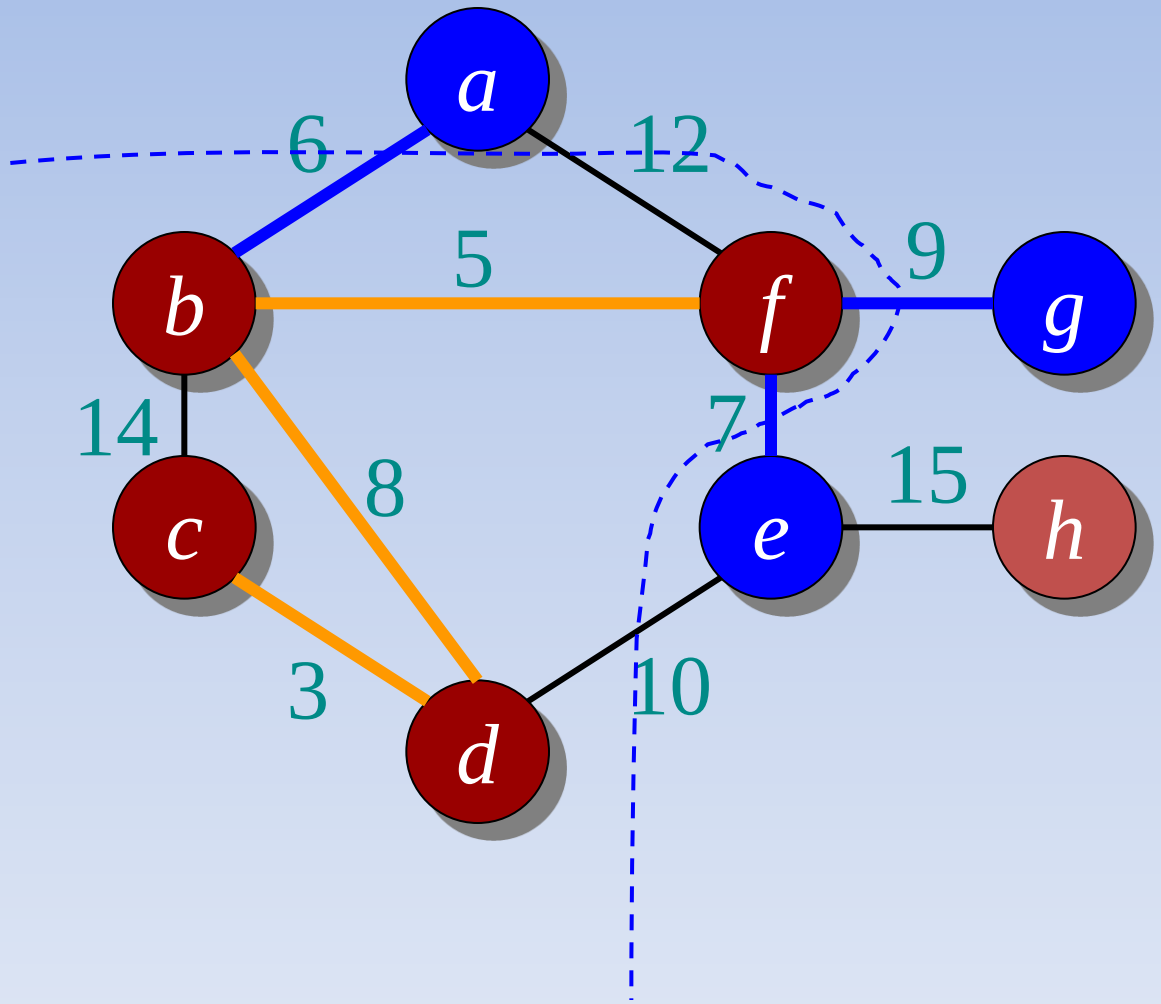
Example



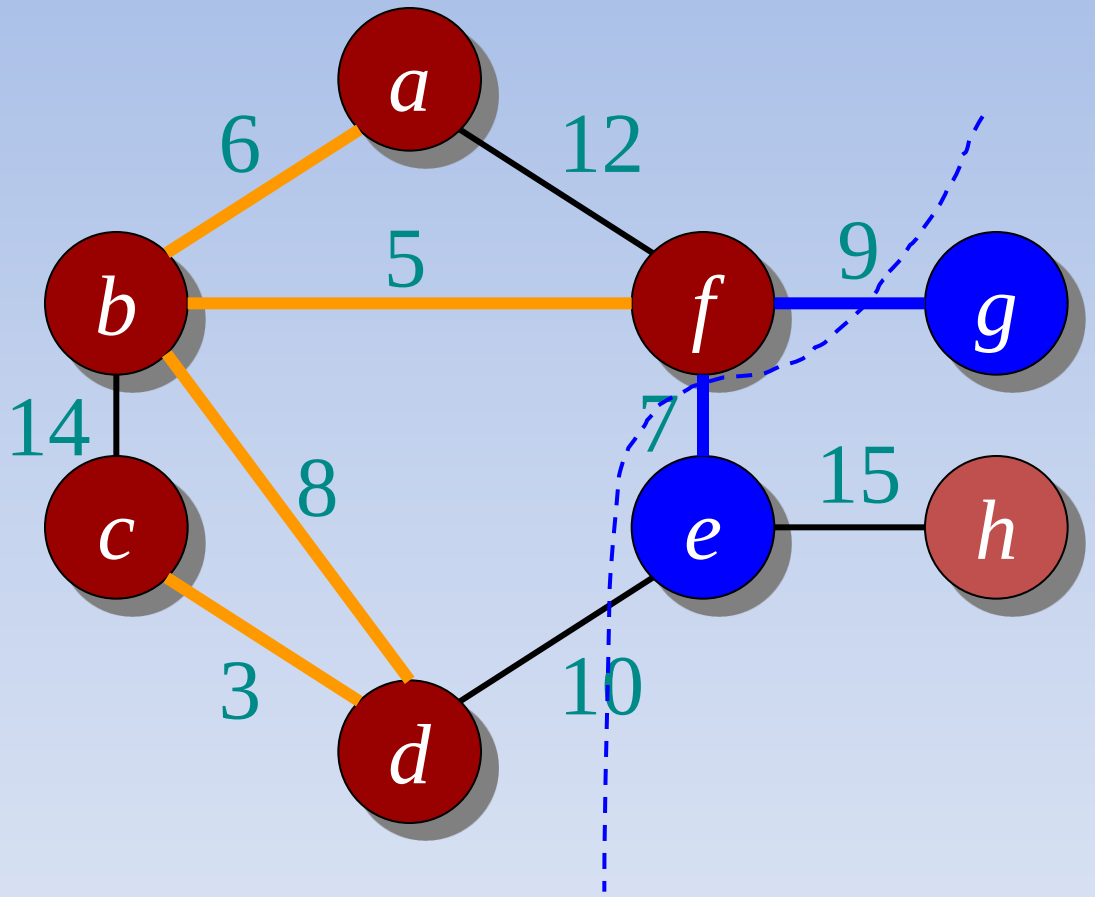
Example



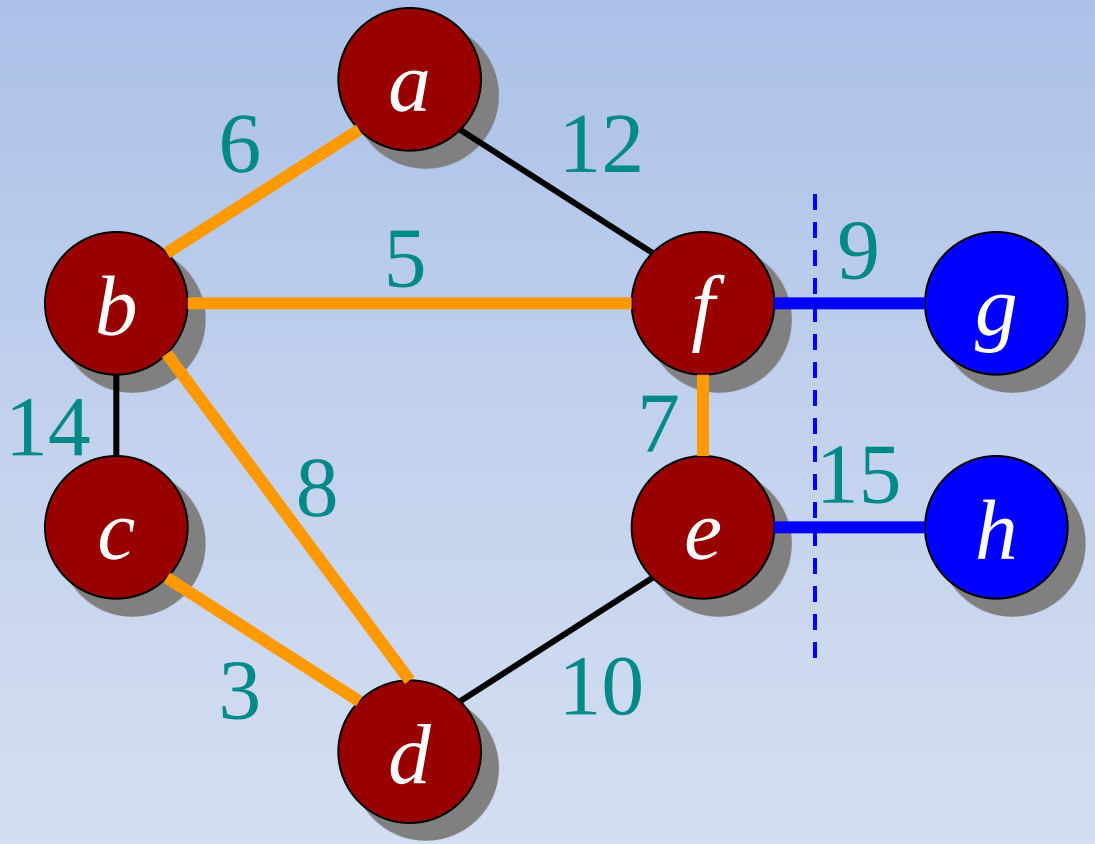
Example



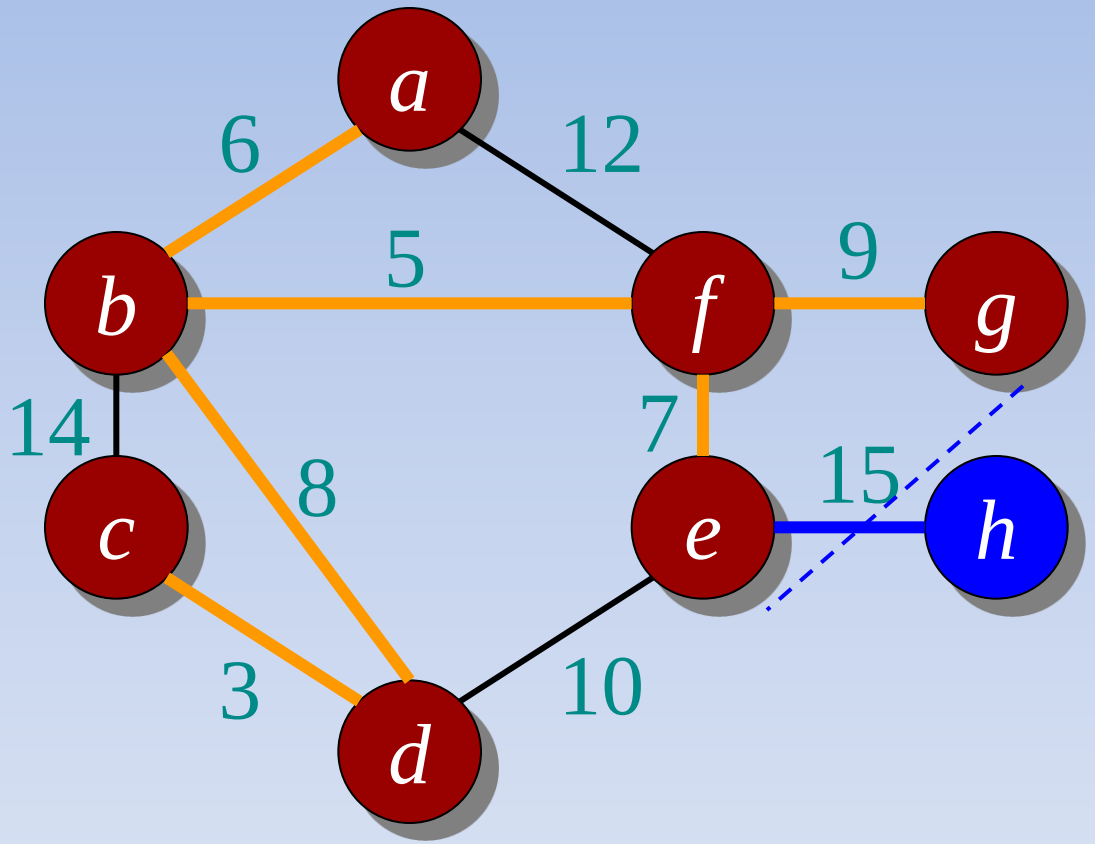
Example



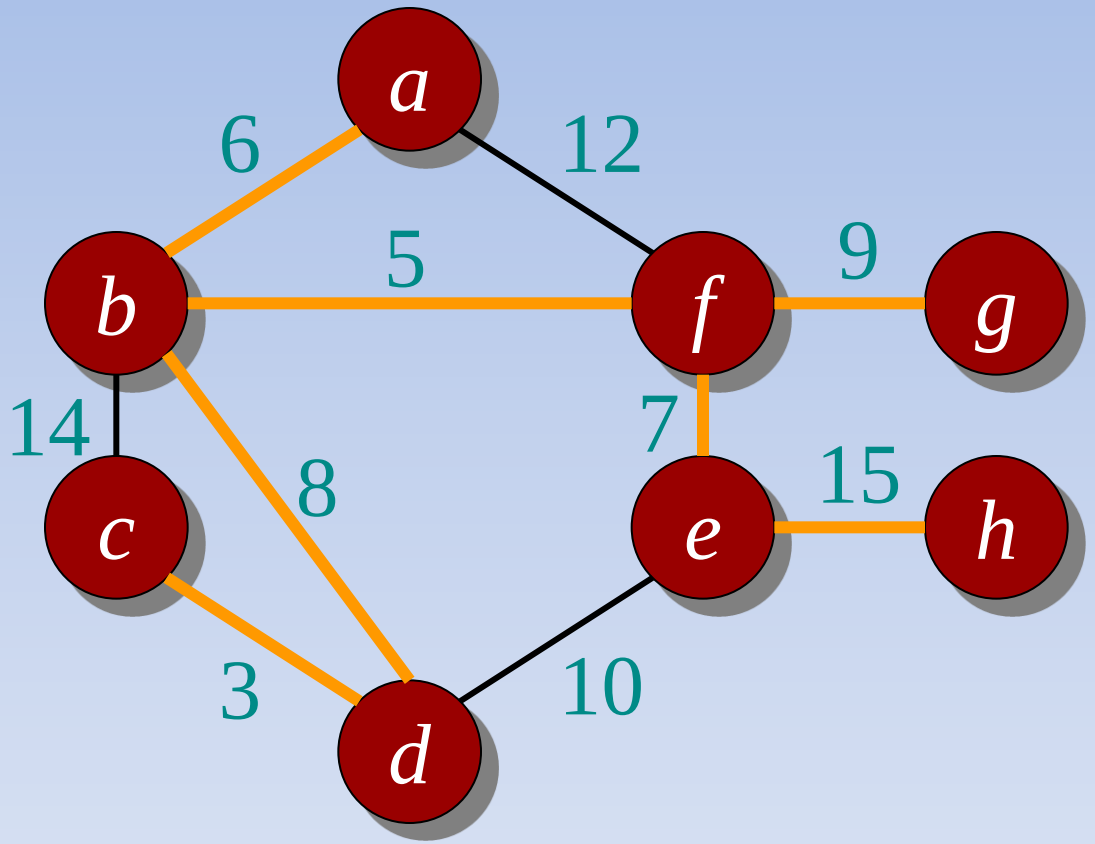
Example



Example



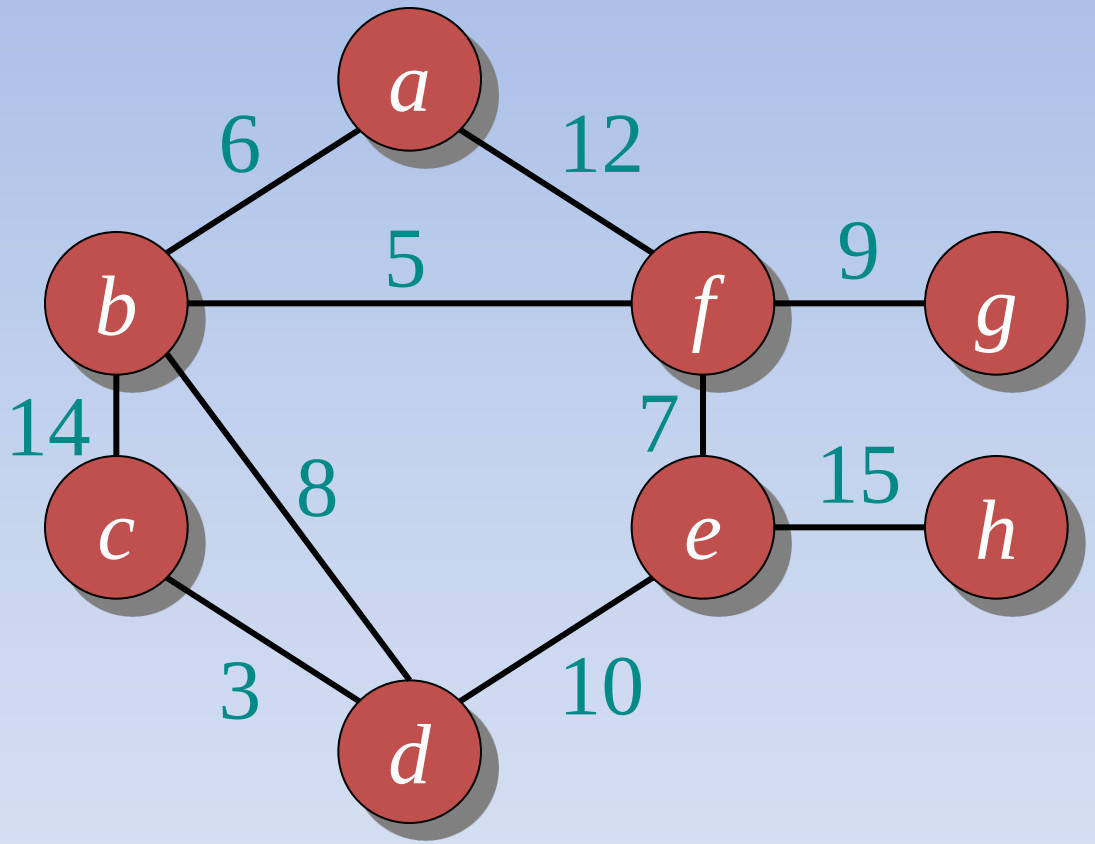
Example



Total weight = 3 + 8 + 6 + 5 + 7 + 9 + 15 = 53

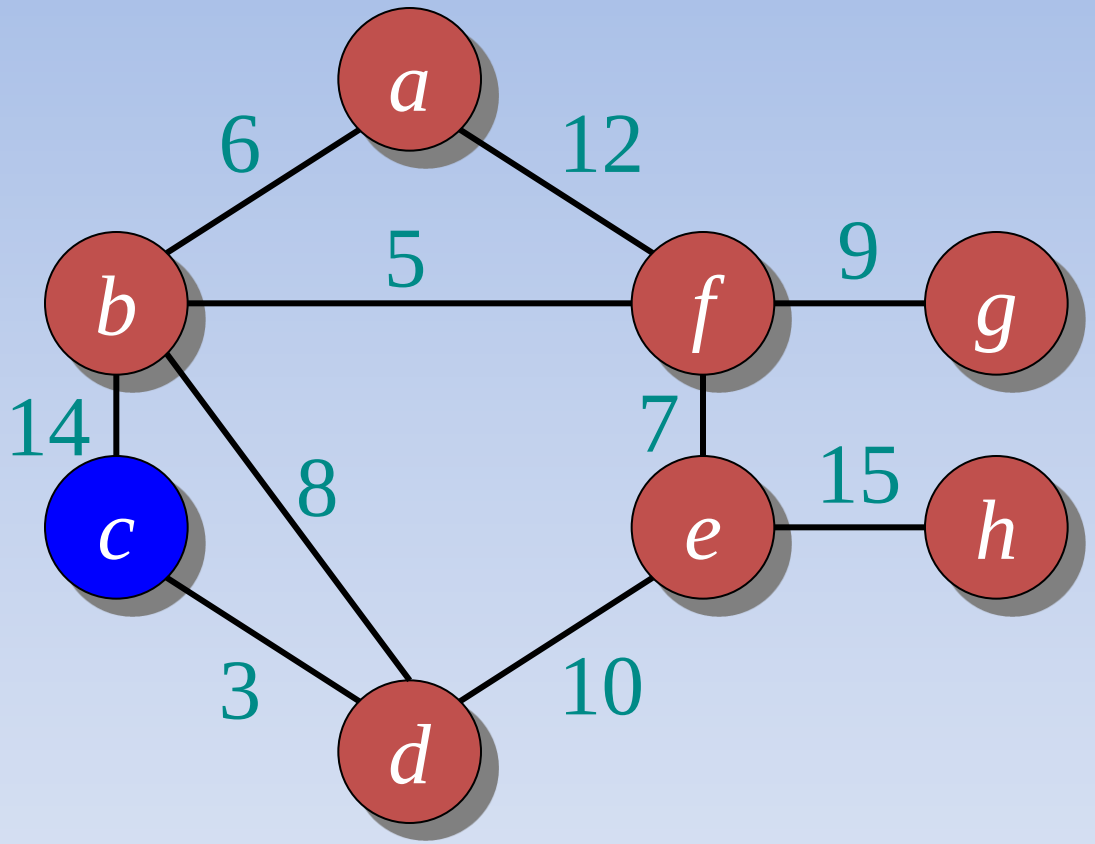
Prim Example w/ Priority Queue

Example



a	b	c	d	e	f	g	h
∞	∞	∞	∞	∞	∞	∞	∞

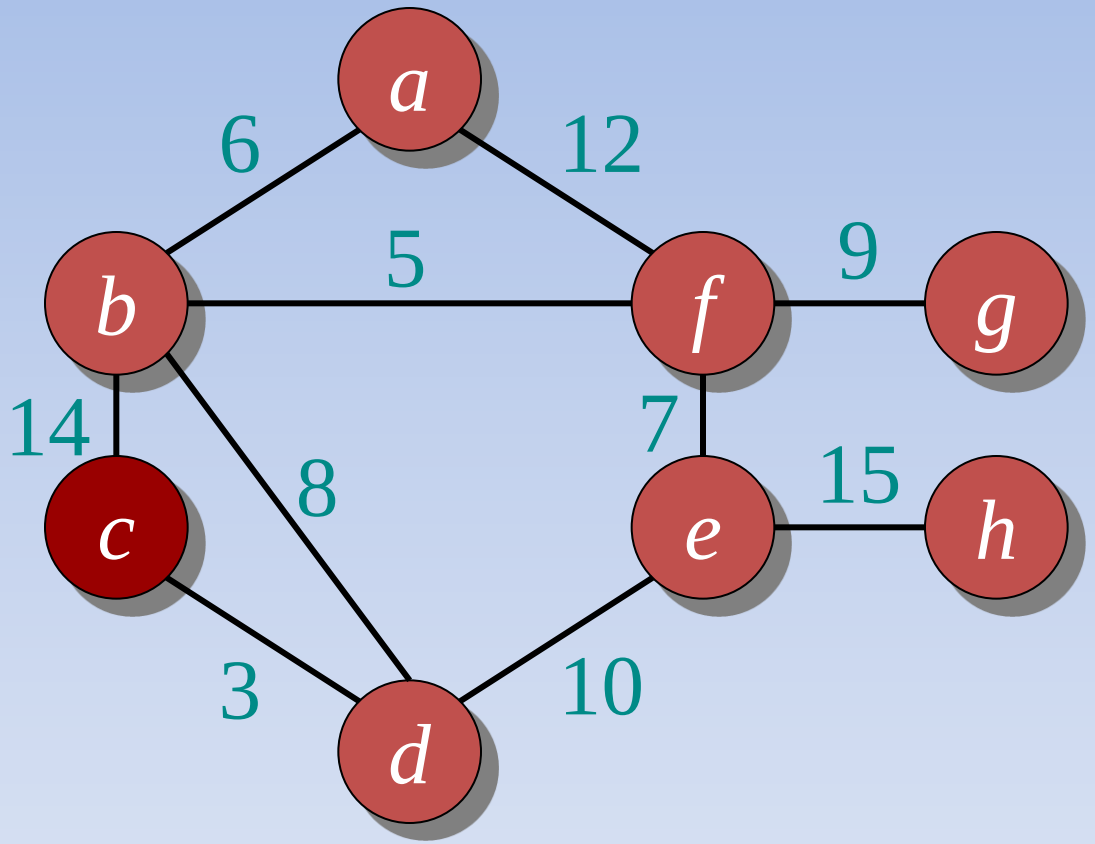
Example



ChangeKey

c	b	a	d	e	f	g	h
0	∞	∞	∞	∞	∞	∞	∞

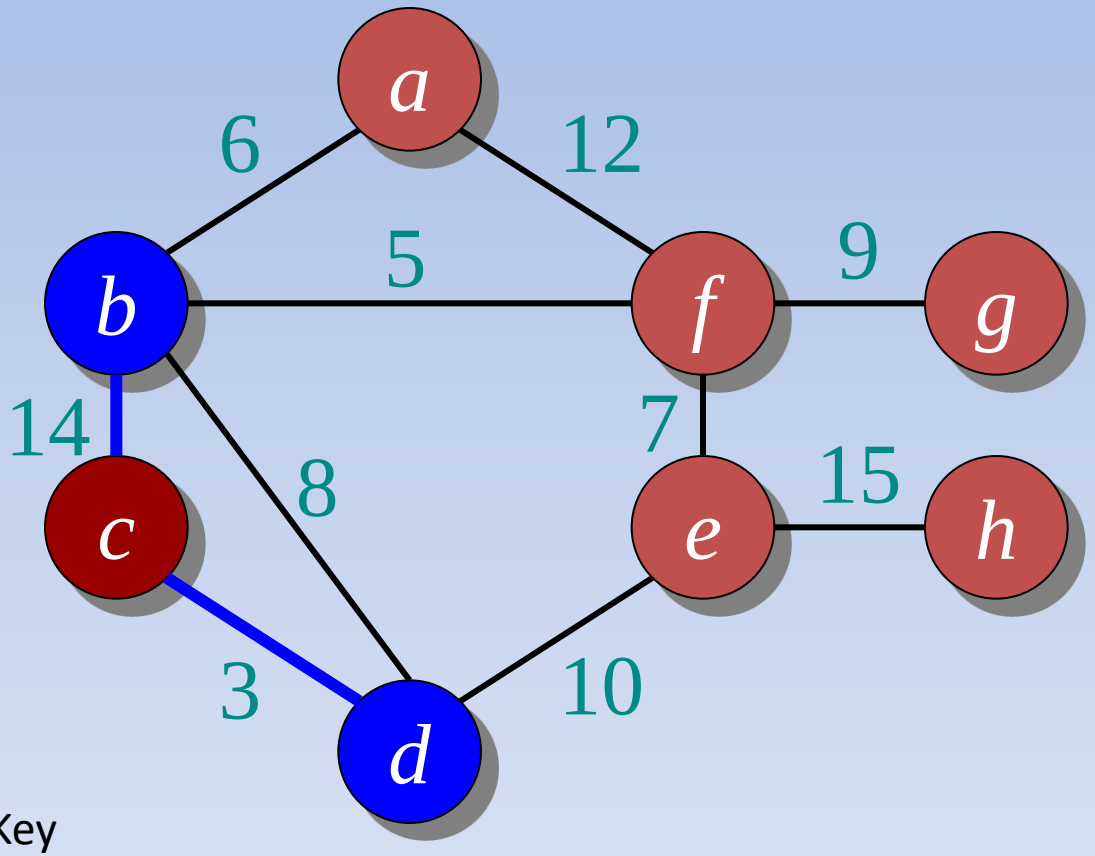
Example



ExctractMin

h	b	a	d	e	f	g
∞	∞	∞	∞	∞	∞	∞

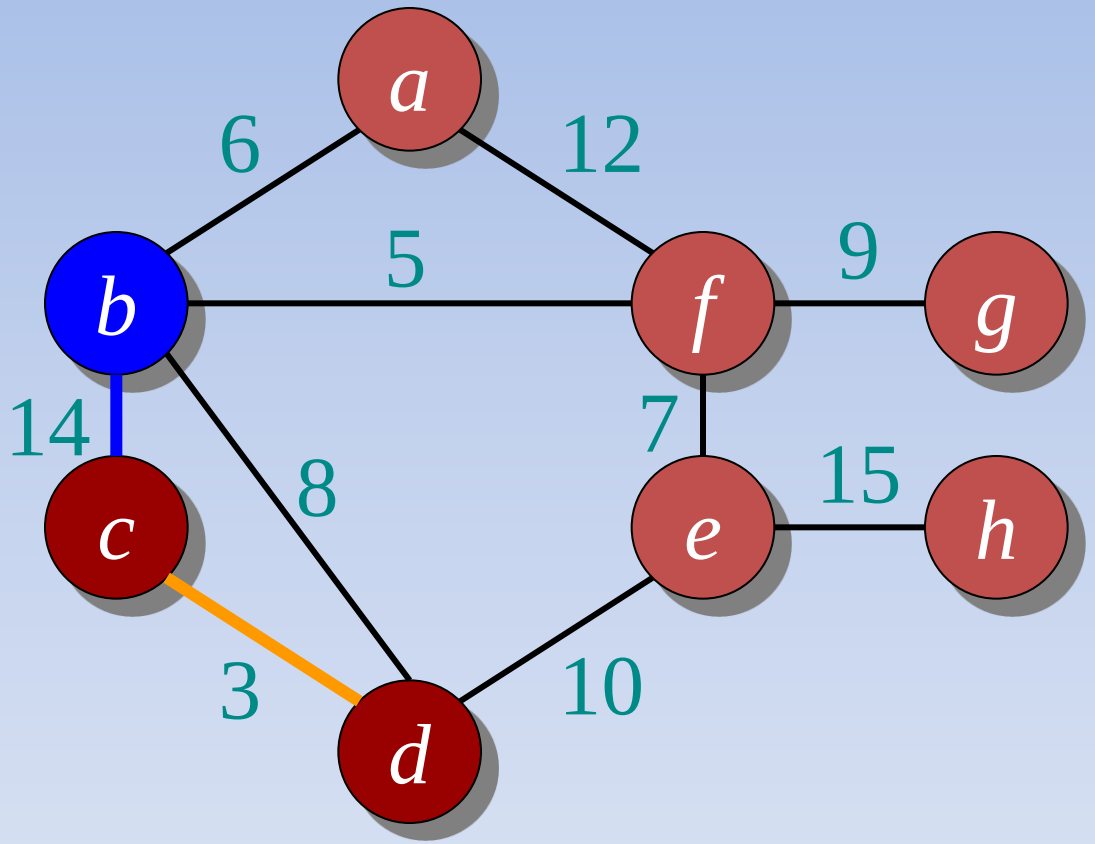
Example



ChangeKey

d	b	a	h	e	f	g
3	14	∞	∞	∞	∞	∞

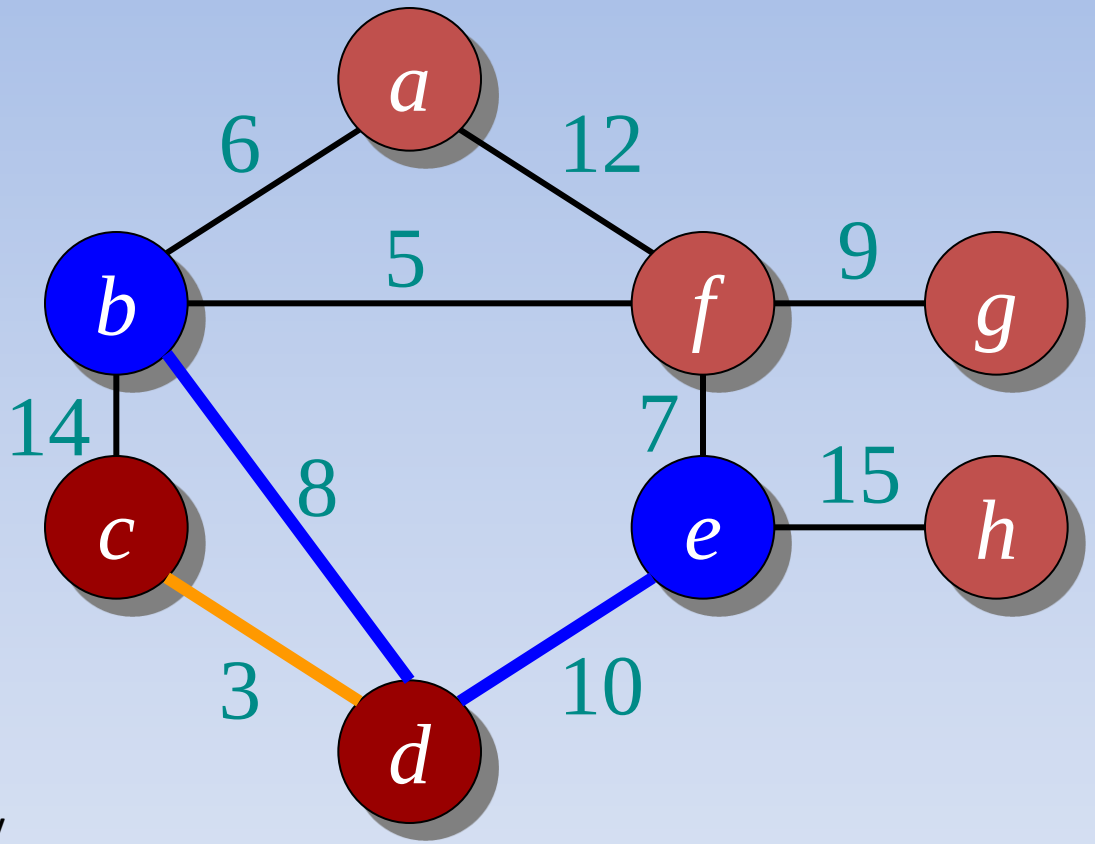
Example



ExctractMin

b	g	a	h	e	f
14	∞	∞	∞	∞	∞

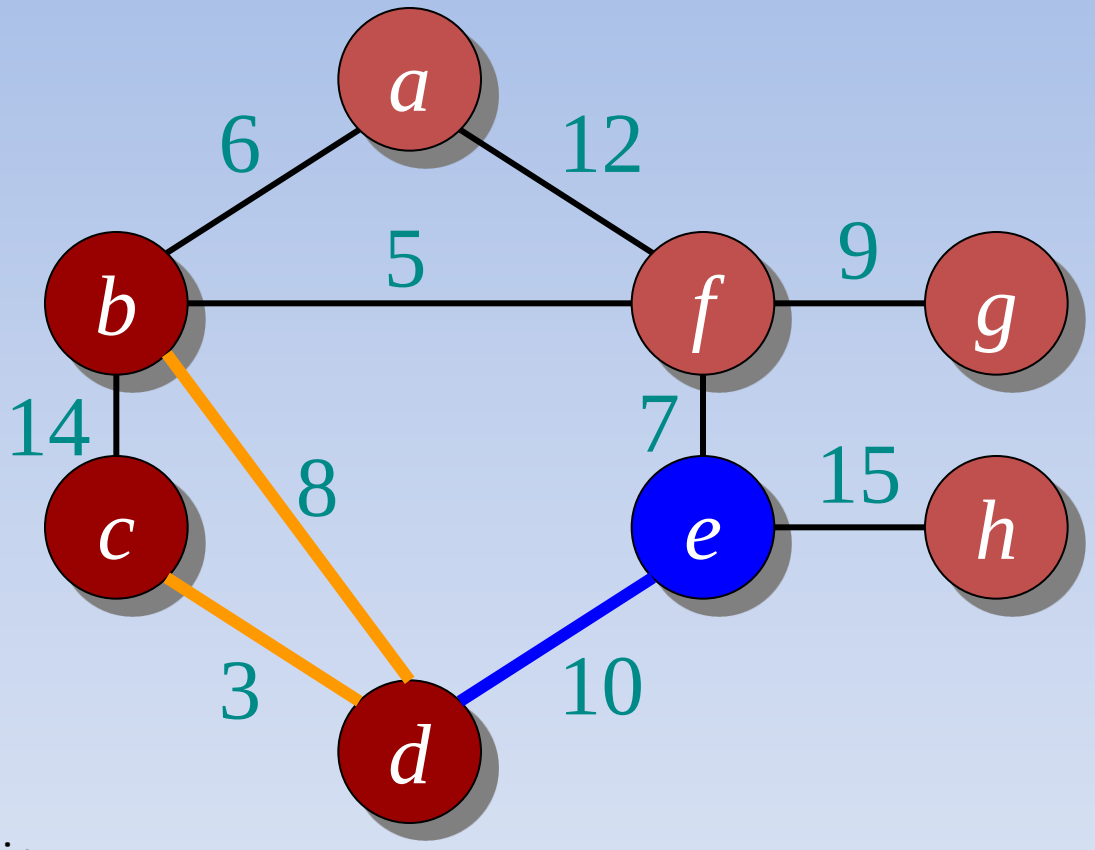
Example



Changekey

b	e	a	h	g	f
8	10	∞	∞	∞	∞

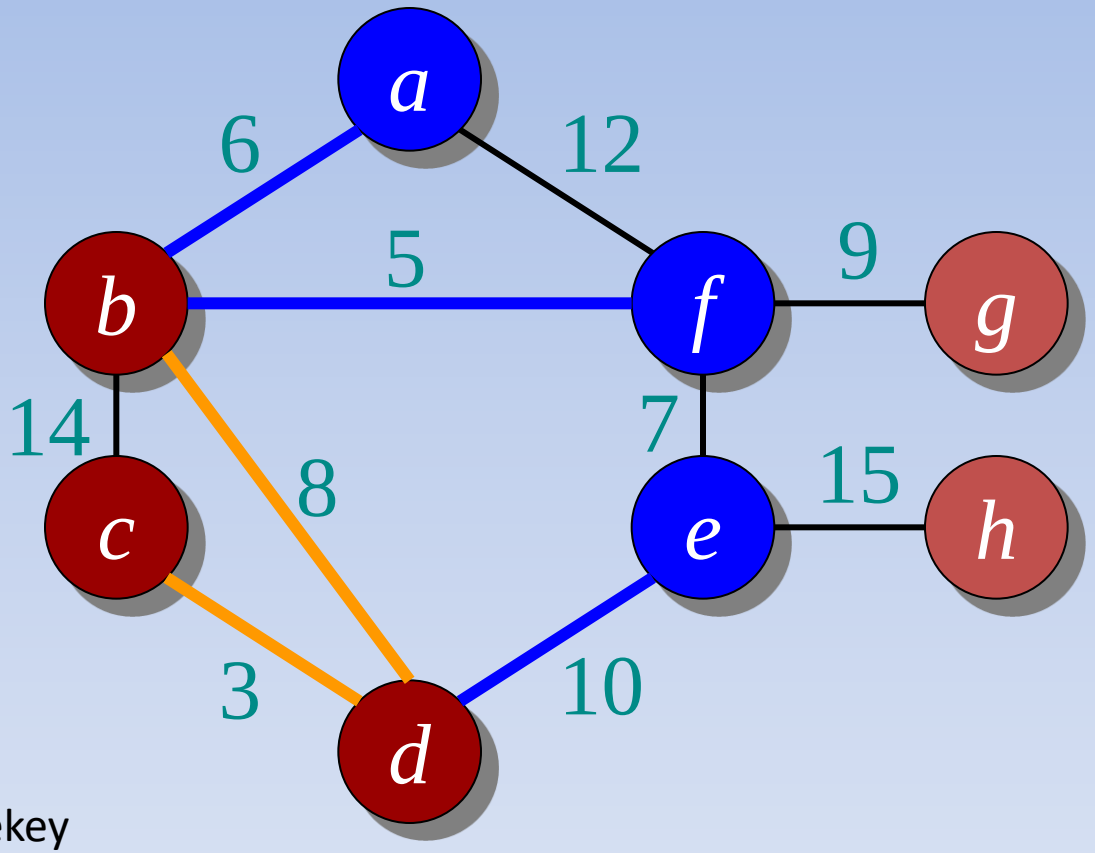
Example



ExtractMin

e	f	a	h	g
10	∞	∞	∞	∞

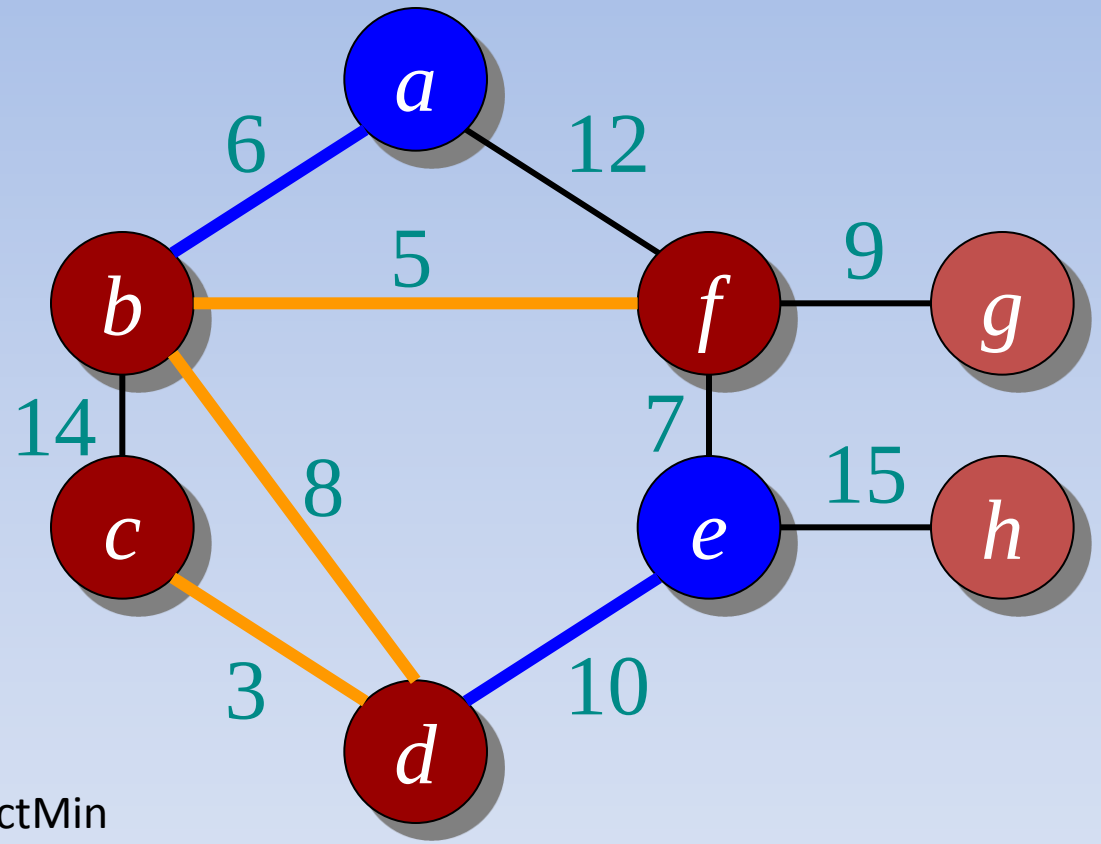
Example



Changekey

f	e	a	h	g
5	10	6	∞	∞

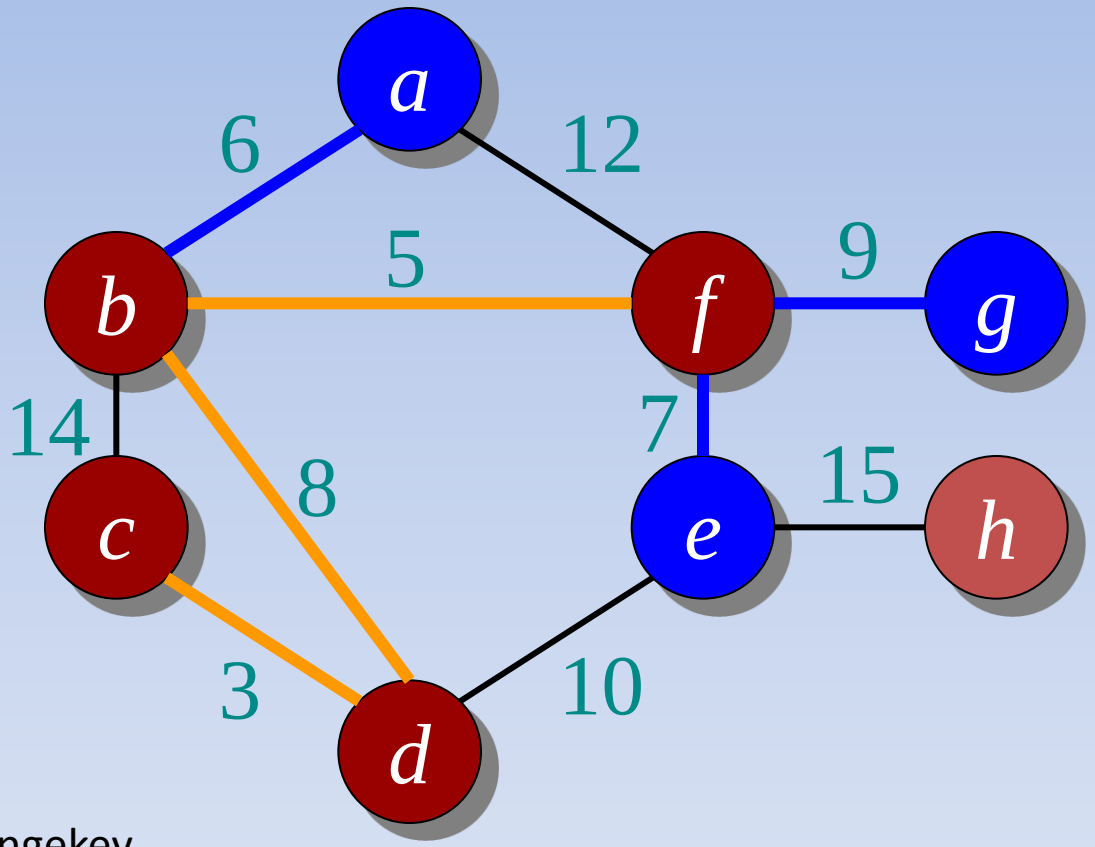
Example



ExtractMin

a	e	g	h
6	10	∞	∞

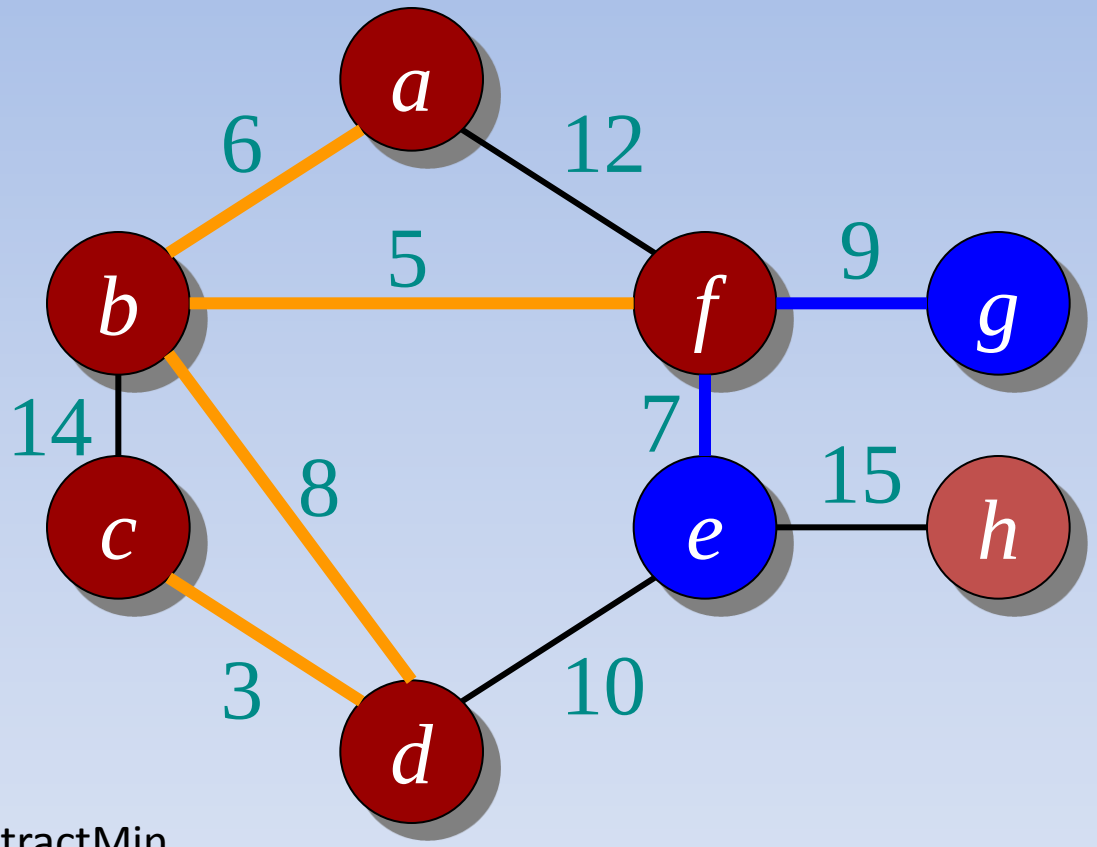
Example



Changekey

a	e	g	h
6	7	9	∞

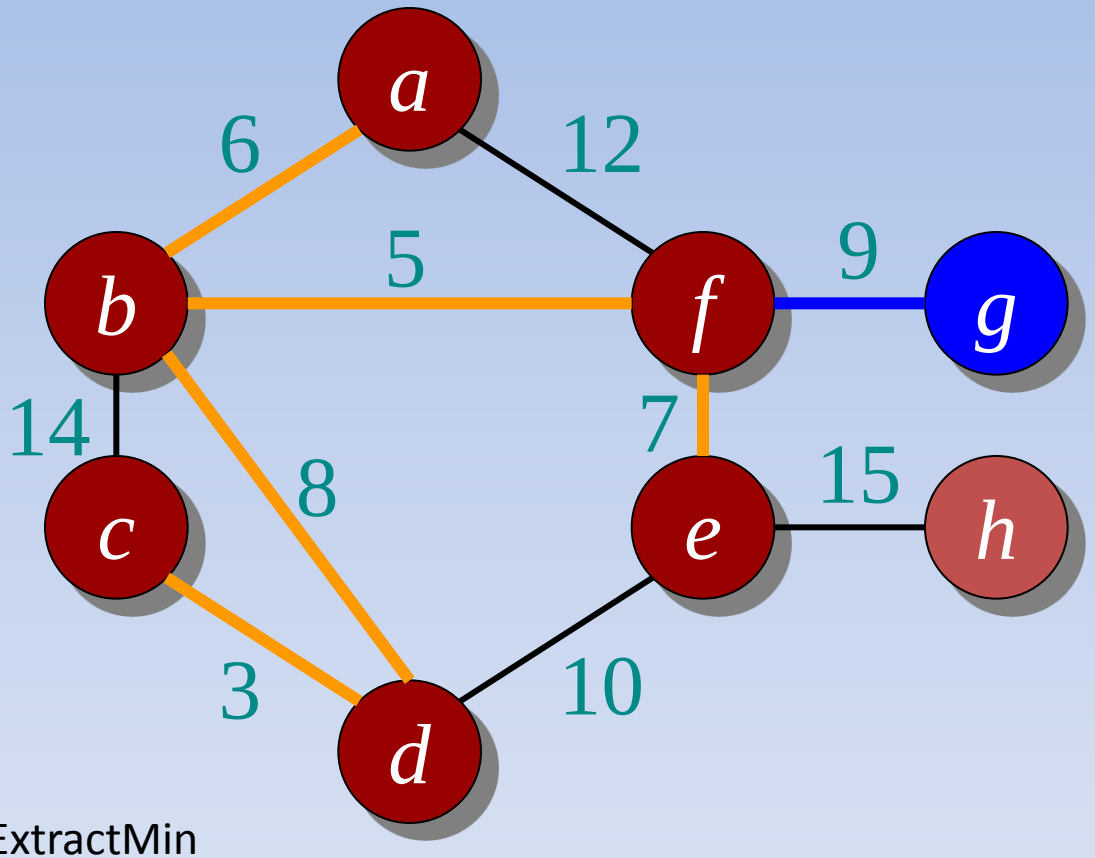
Example



ExtractMin

e	h	g
7	∞	9

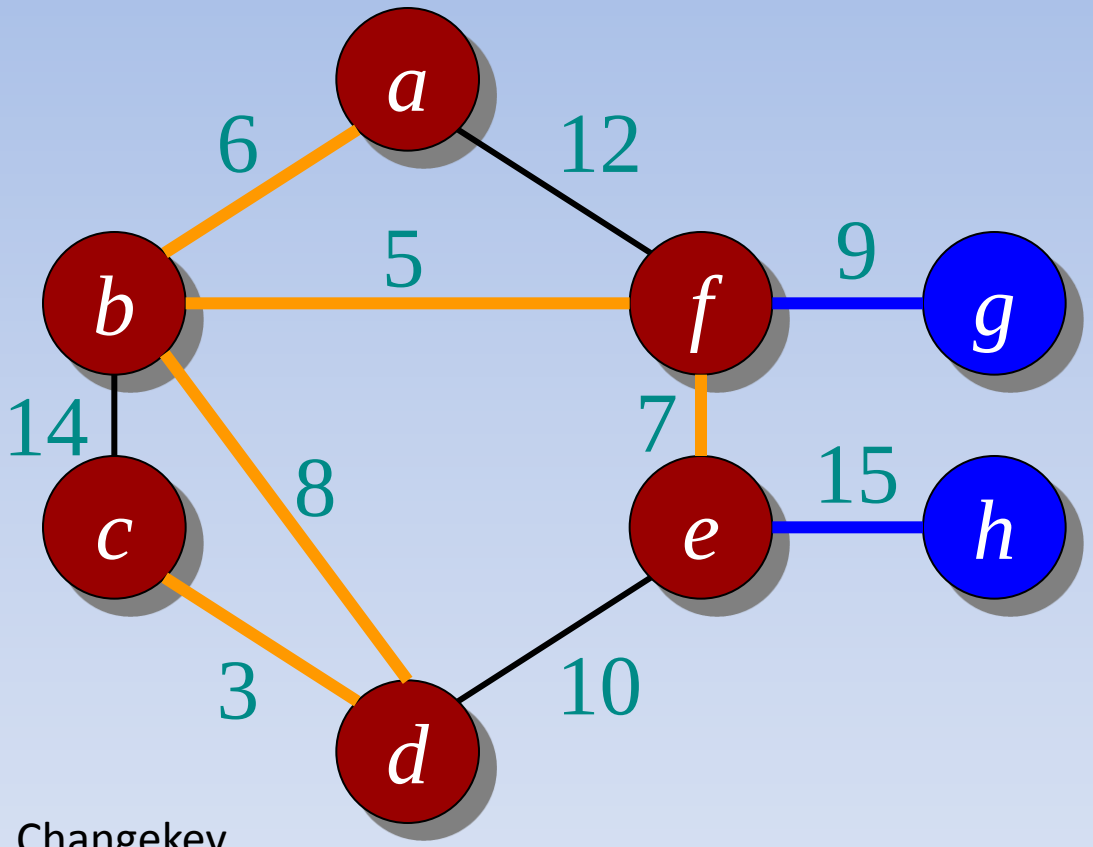
Example



ExtractMin

g	h
9	∞

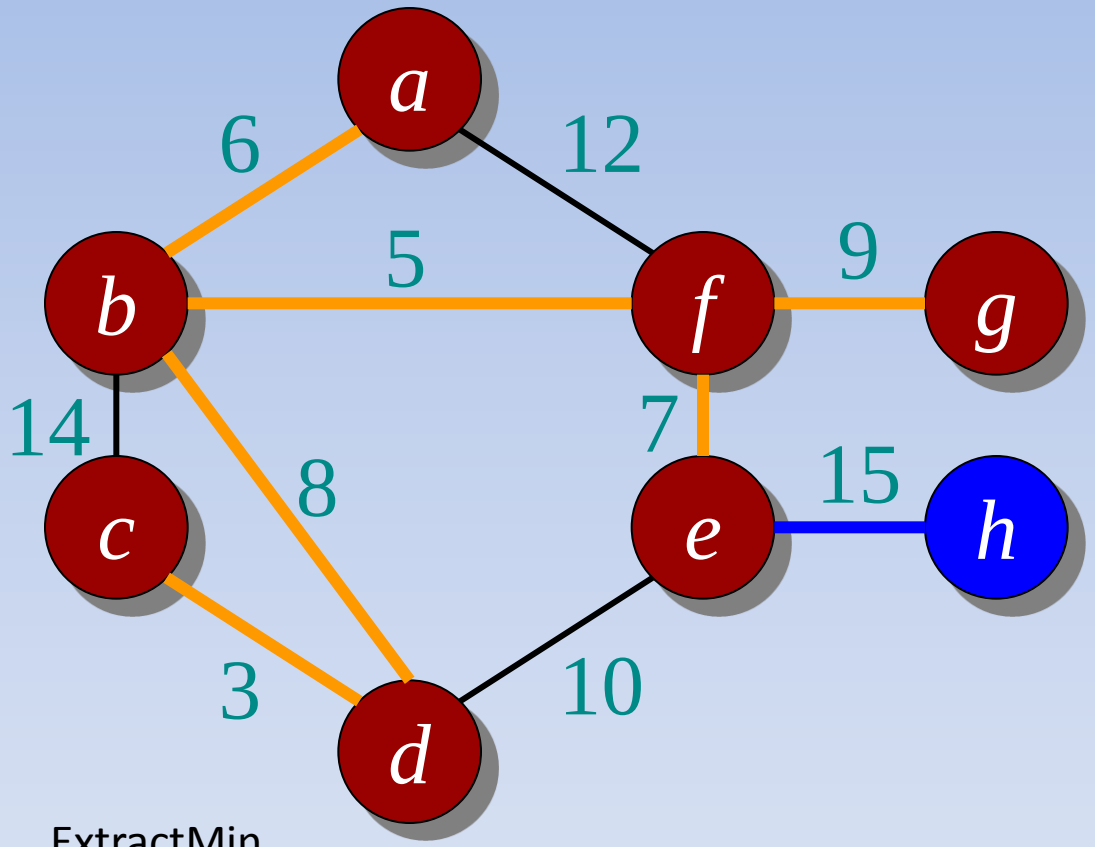
Example



Changekey

g	h
9	15

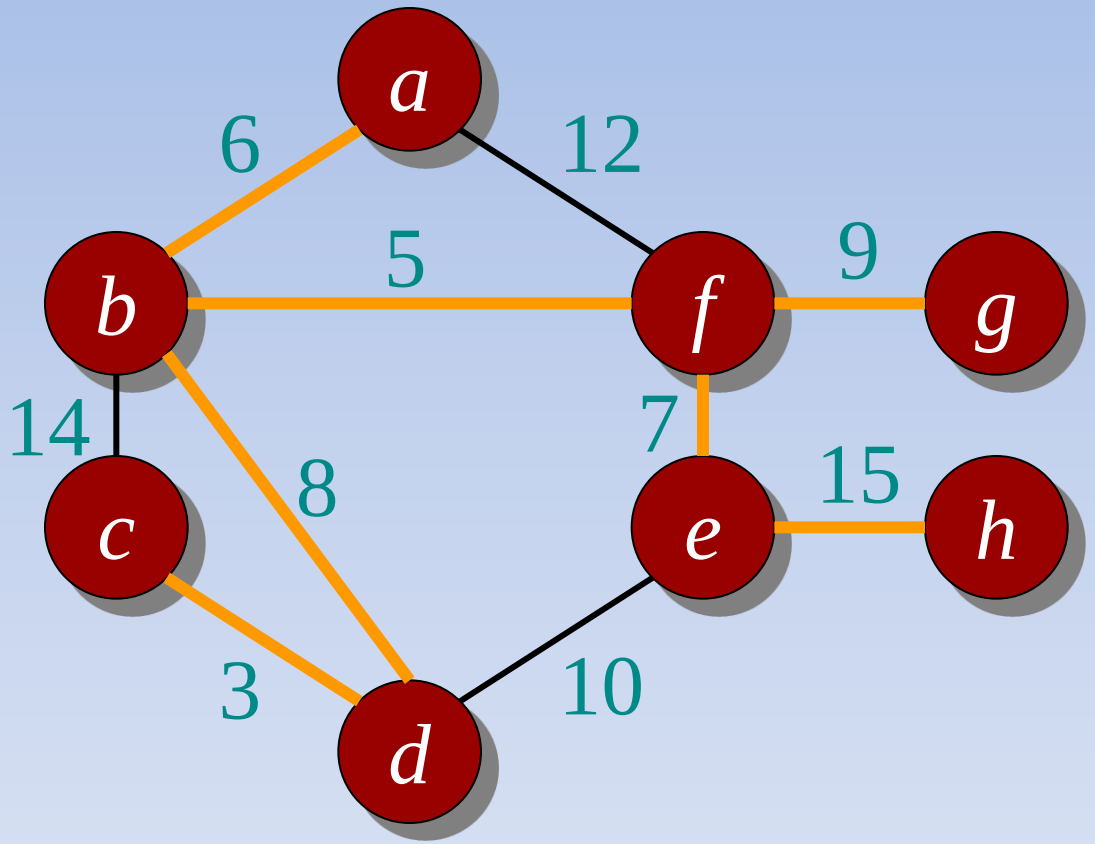
Example



ExtractMin

h
15

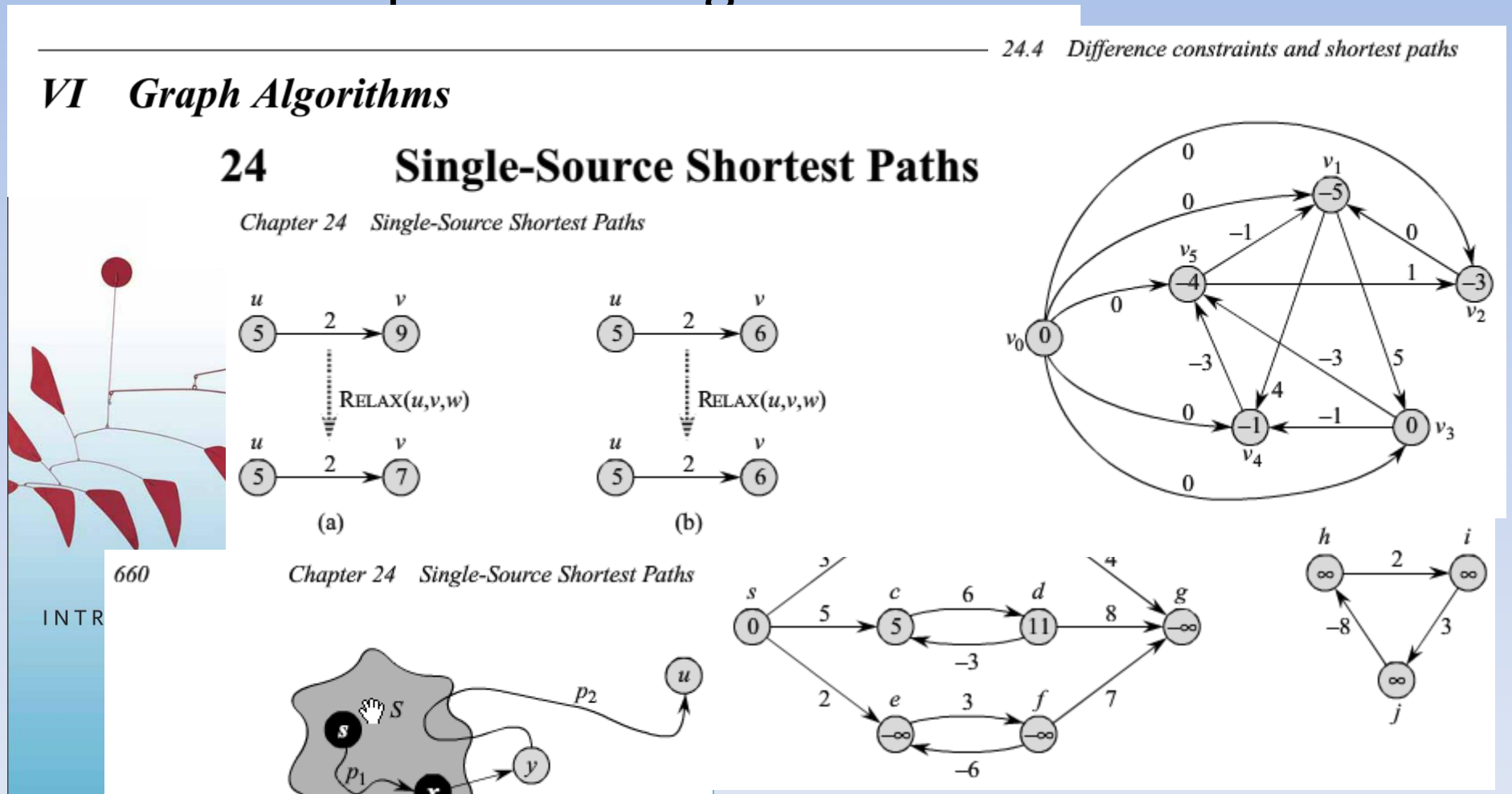
Example



Design and Analysis of Algorithms

Section VI : Graph Algorithms

Chapter 24: Single-Source Shortest Paths

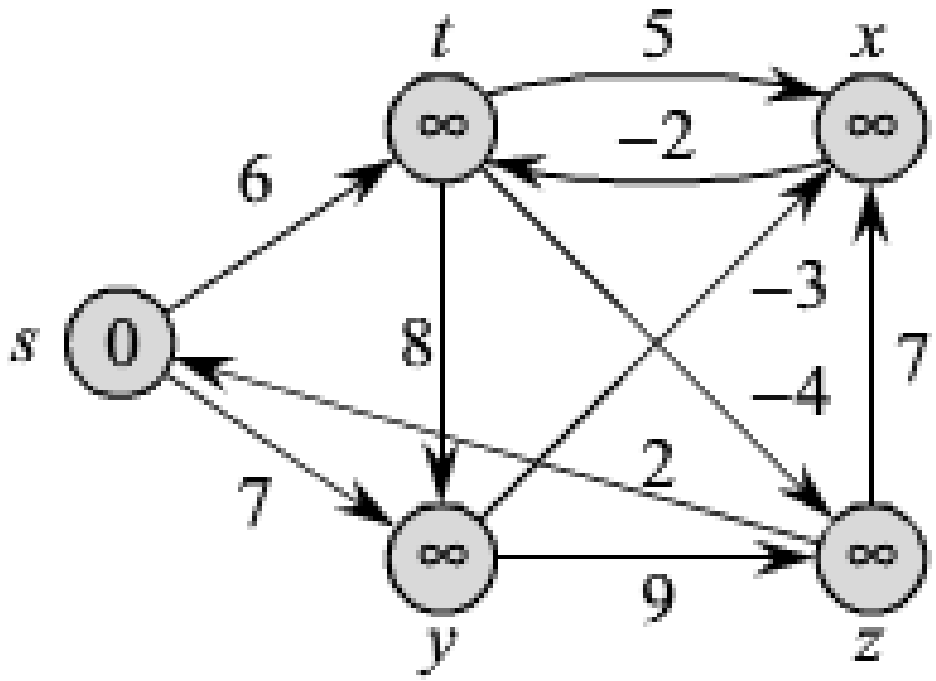


Bellman-Ford

- Solves Single-Source Shortest-Paths Problem in General Case.
  - Weights may be negative.
- Given Graph G=(V,E) with source s and weight function w returns:
  - False: if negative-weight cycle exist
  - True: Otherwise

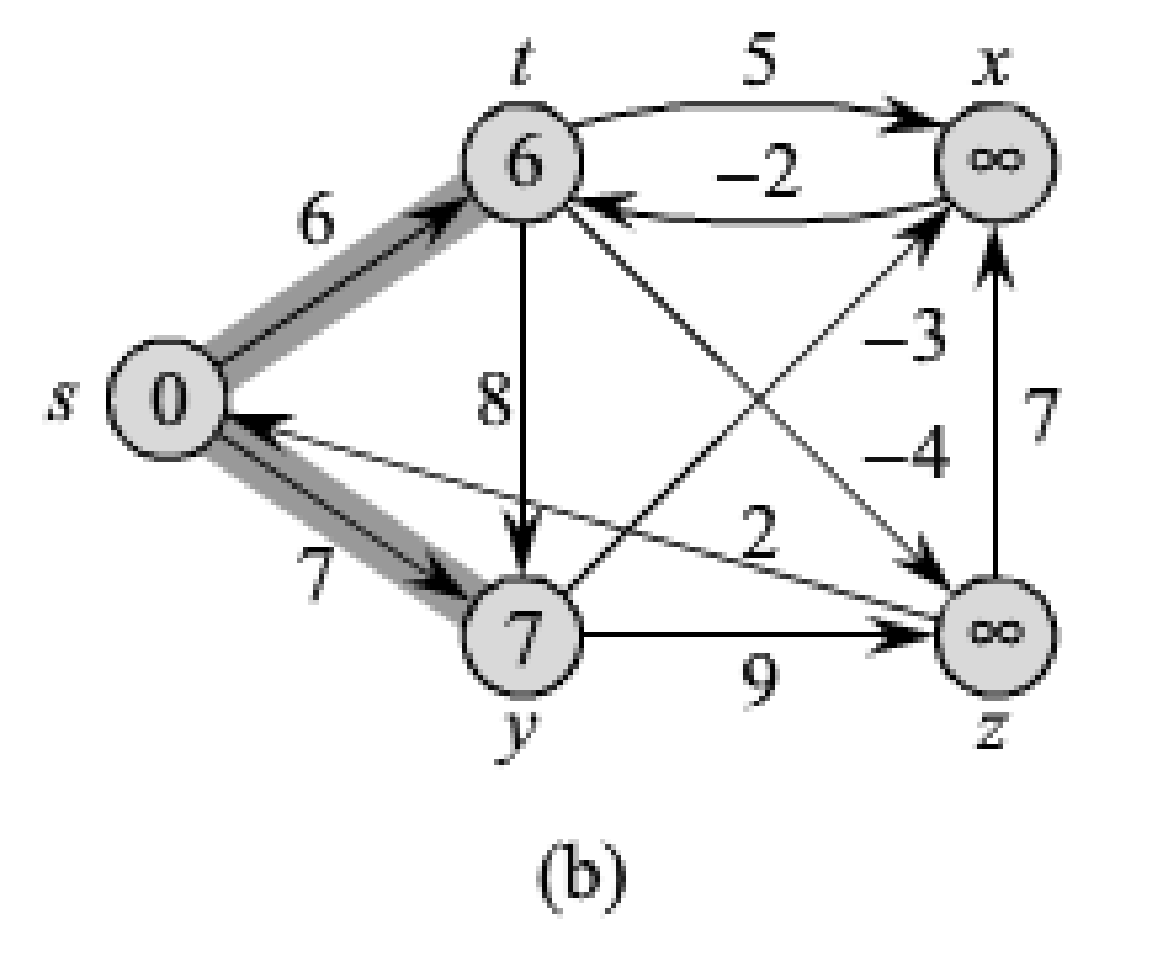
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Bellman-Ford Example



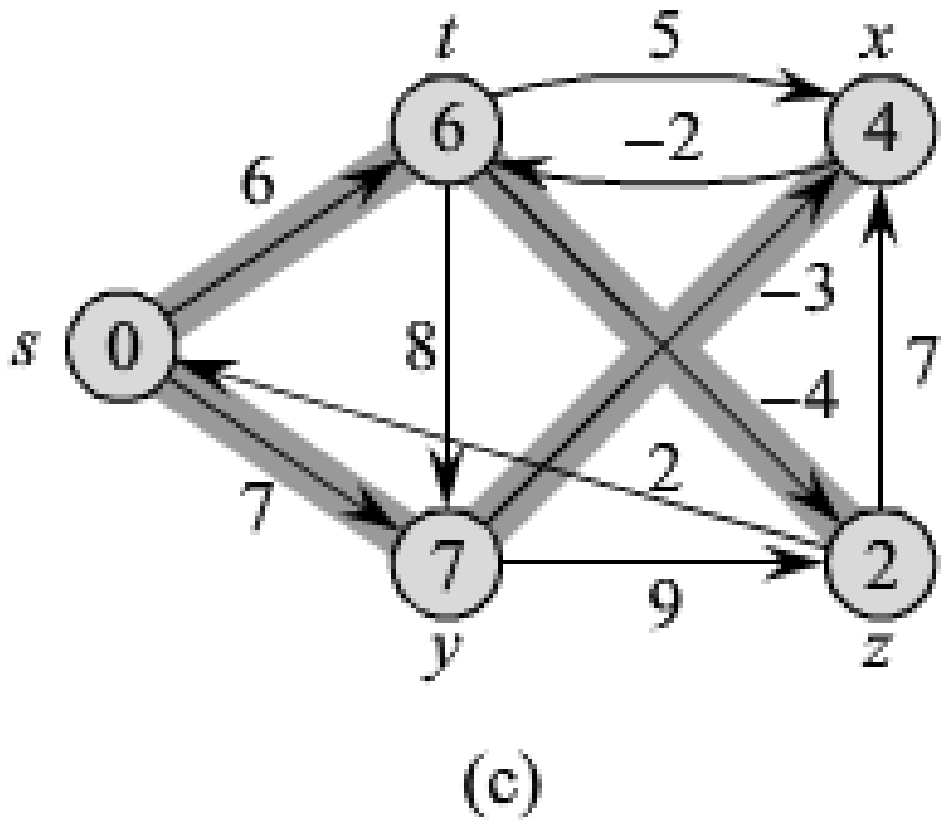
(a)

Bellman-Ford Example



- First Iteration: t & y relaxed

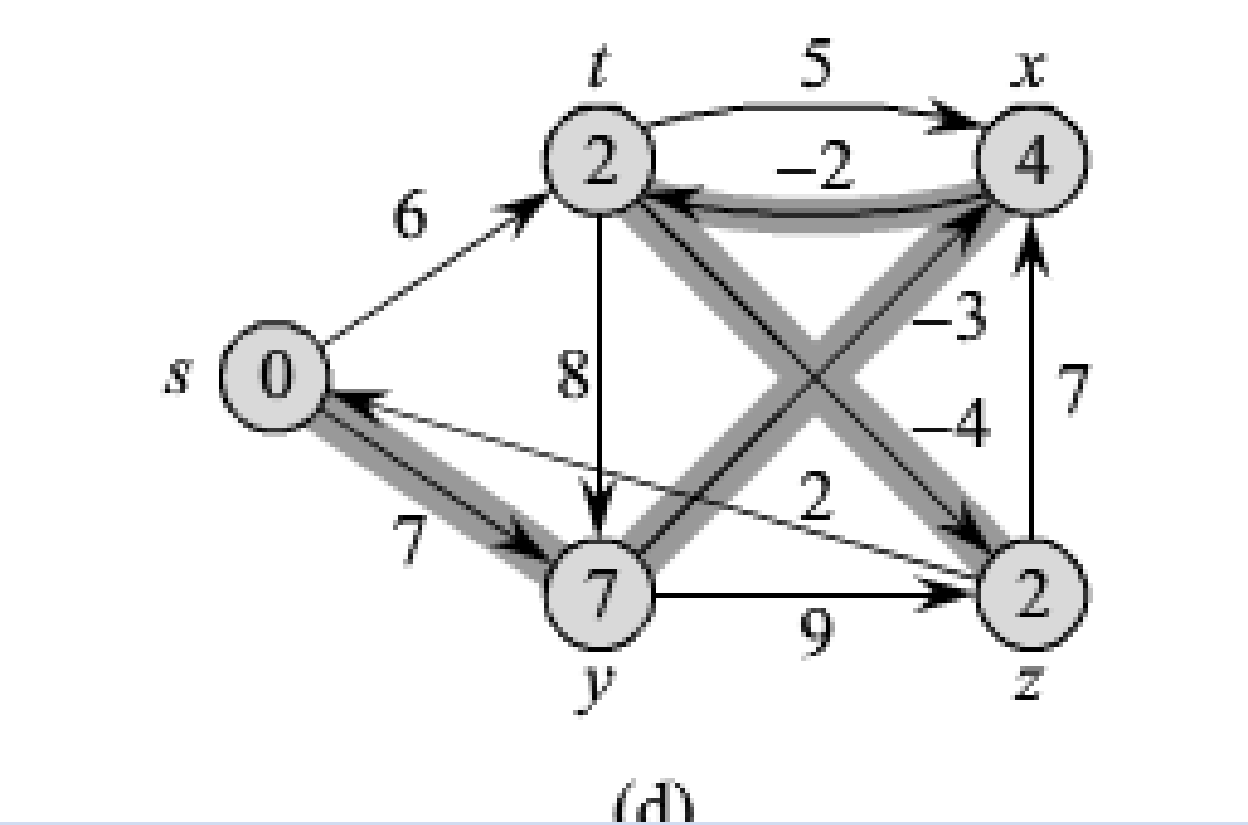
Bellman-Ford Example



(c)

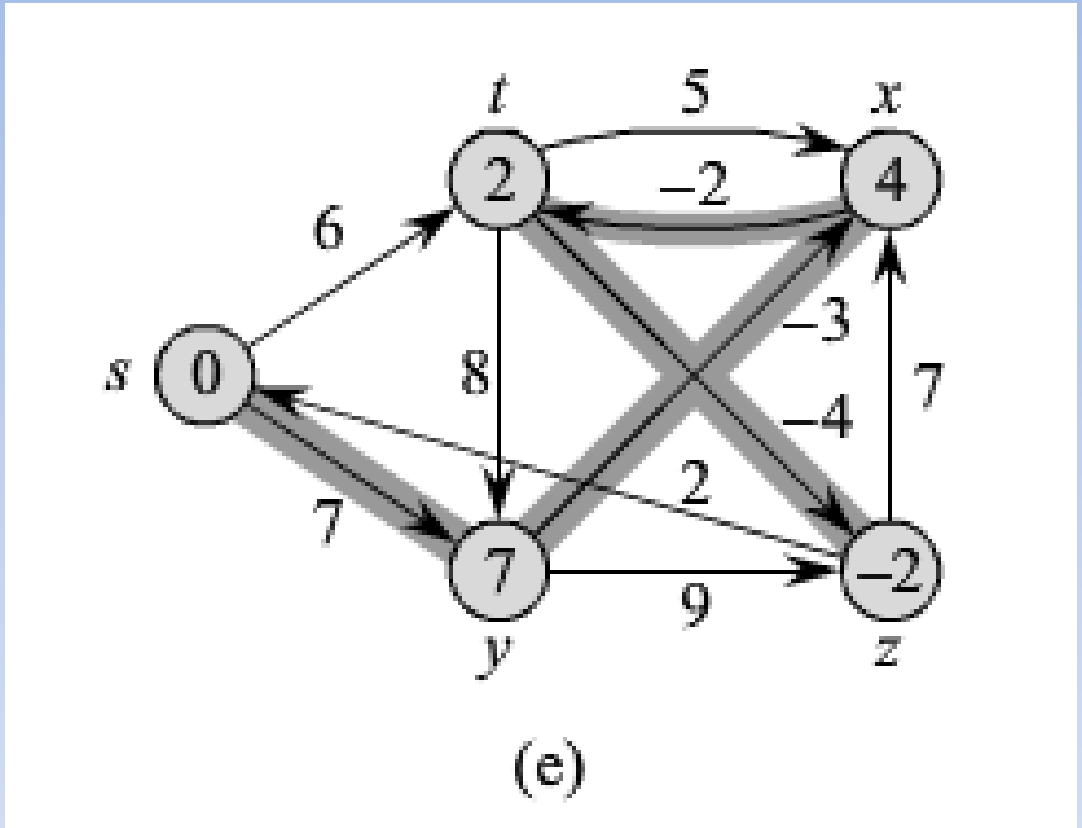
- Second Iteration: x & z relaxed

Bellman-Ford Example



- Third Iteration: t relaxed
  - **Via: s, y, x, t**

Bellman-Ford Example



- Fourth Iteration: z relaxed
  - **Via: s, y, x, z**

Bellman-Ford Complexity

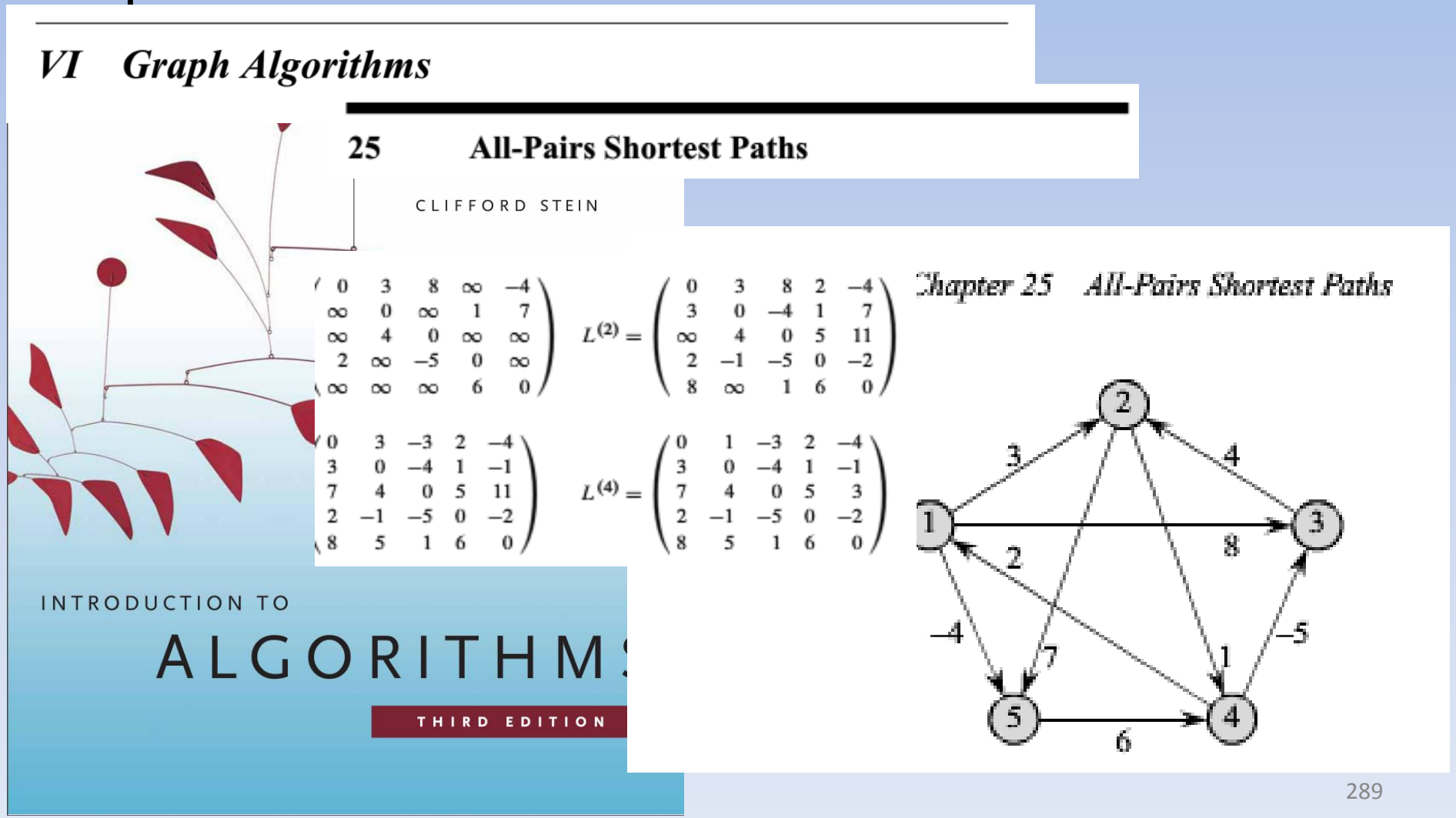
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

- Lines 2-4: |V|-1 passes over Edges in E
- O(VE)

Design and Analysis of Algorithms

Section VI : Graph Algorithms

Chapter 25: All-Pairs Shortest Paths



Floyd-Warshall

- Assume no negative-weight cycles
- Use a different dynamic-programming formulation

Shortest Path Structure: Intermediate Vertex

- $P = \langle v_1, v_2, \ldots, v_i \rangle$
  - Intermediate Vertex is any vertex $\{v_2, \ldots, v_{i-1}\}$
  - Does not include start/end v_1, v_i
- Given that we are using vertices $\{1, 2, \ldots, n\}$
  - There is a linear ordering of our vertices
- Consider a subset of these vertices less than some value k.
- Explore paths using only intermediate vertices $v<k$
  - Iteratively Expand k
  - Construct paths using $v<k$ from those using $v<k-1$

Shortest Path Structure: Intermediate Vertex

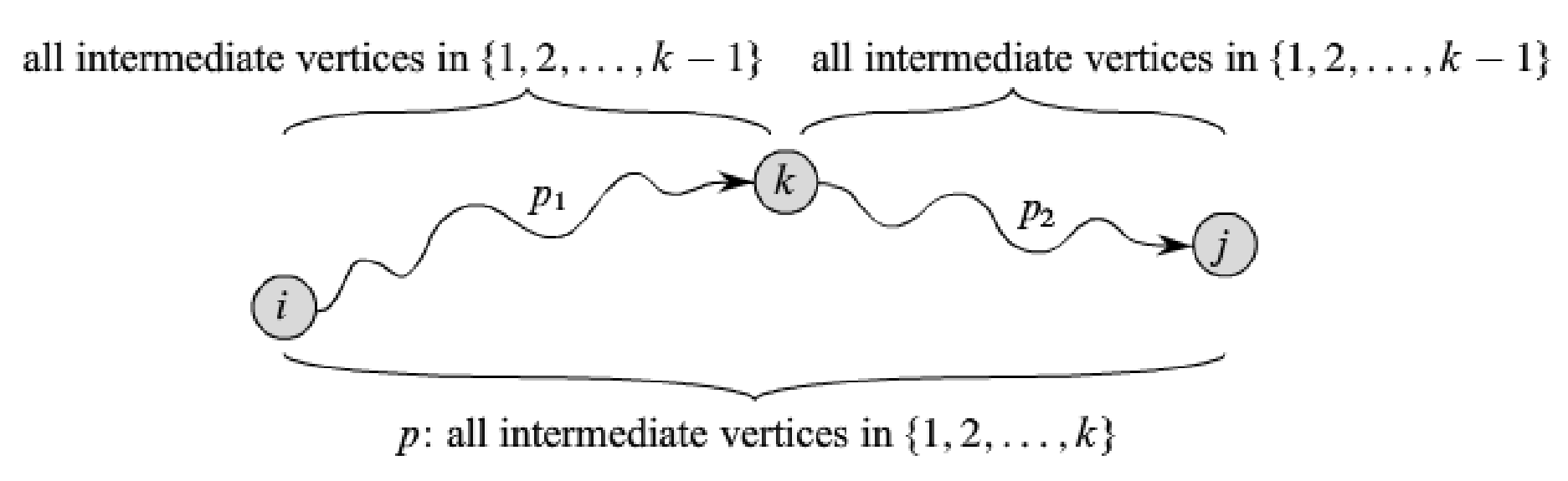
- Then consider a subset of vertices < k
  - {1, 2, 3, …, k}
- Then consider pairs of vertices i, j
  - consider all paths between i,j using only intermediate v in {1, 2, … , k}
  - Let p be minimum-weight path from previous set.
- Floyd-Warshall uses relationship between:
  - minimum-weight path using v<k-1
  - and that using v<k.

Shortest Path Structure First Case

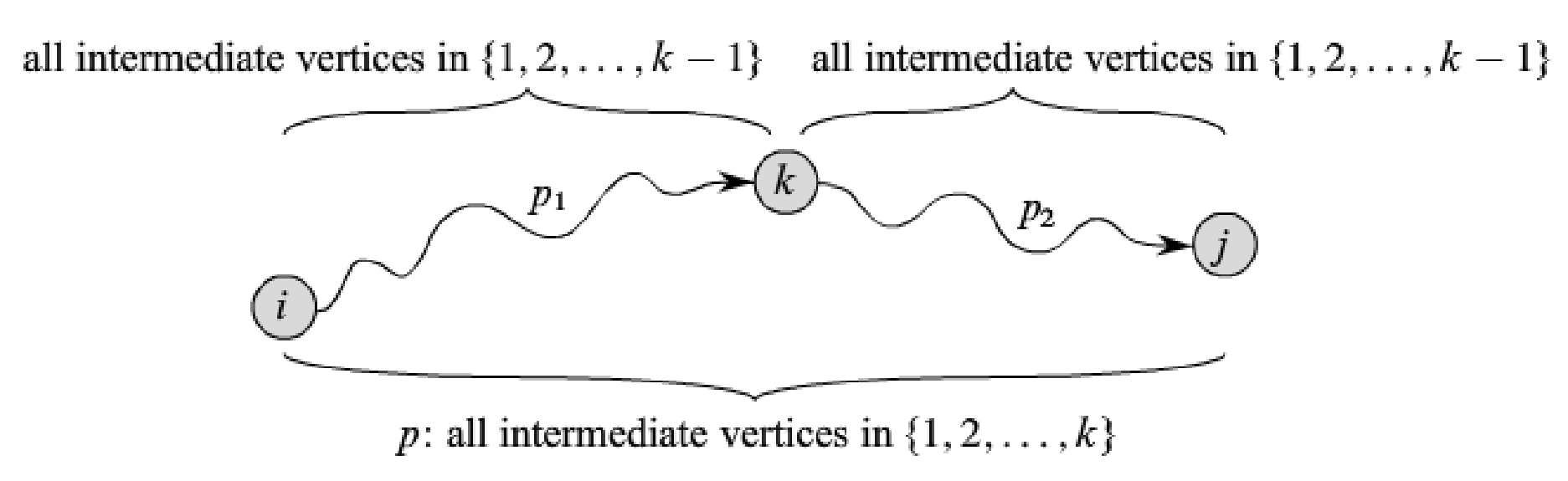
- Then consider:
  - subset of vertices $v < k-1 = \{1, 2, 3, \ldots, k-1\}$
  - Let p be minimum-weight path between i,j with intermediate v only from $\{1, 2, 3, \ldots, k-1\}$
- Then q the minimum path with intermediate v only in $\{1, 2, 3, \ldots, k\}$ has property:
  - k **is not** intermediate of q
    - Then the shortest path from k-1 case is same as case with k
    - Then all intermediate vertices must be only in $\{1, 2, \ldots, k-1\}$

Shortest Path Structure
Second Case

- k **is** intermediate of q (our new shortest path) then we can break it into two pieces
  - k is an intermediate node, and there are no cycles so it must break q into two parts: i $-p_1$-> k $-p_2$->j
  - Both p_1 and p_2 must have all of their vertices v in {1, 2, …, k-1}
  - p_1 and p_2 must both be shortest paths from there start and finish vertices using only v < k-1.



- Now we want to exploit this and develop recursive solution.



- Now we want to exploit this and develop recursive solution.
- Develop matrix:
  - $d_{ij}^{(k)}$ = the weight of a shortest path from i to j using only intermediate v<k
  - $d_{ij}^{(0)}$ = No intermediate v means edge from i to j.
  - $d_{ij}^{(0)} = w_{ij}$

$d_{ij}^{(k)}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases} \tag{25.5}$$

$d_{ij}^{(k)}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases} \tag{25.5}$$

- $d_{ij}^{(n)}$ will have out shortest path info.

```
FLOYD-WARSHALL(W)
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

Johnson's Algorithm

- For Sparse Graphs – Better than Floyd-Warshall or Repeated Matrix Squaring
- $O(V^2 \lg V + VE)$

- Returns a matrix of shortest-path weights OR Reports Negative Weight Cycle

Johnson's Algorithm w/ Reweighting

- Given Graph G=(V,E) with all edge weights w>=0.
- Dijkstra's shortest path run on each vertex v in V w/ Fibonacci-Heap min-priority queue
  - $O(V^2 \lg V + VE)$
- If G has edge weights less than 0
  - Compute new set of non-negative weights

New Weights: $\widehat{w}$

1. $\forall u, v \in V$, a shortest path between u and v with w must also be a shortest path with $\widehat{w}$
2. $\widehat{w}$ is nonnegative for all edges

Computing New Weights

- Reweight $\widehat{w}$ using function h(v) for v in V.
- $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$

Computing New Weights

- Reweight $\widehat{w}$ using function h(v) for v in V.
- $\widehat{w}$(u, v) = w(u, v) + h(u) – h(v)

- Reweighting doesn't change Shortest Path

Sum
Telescopes

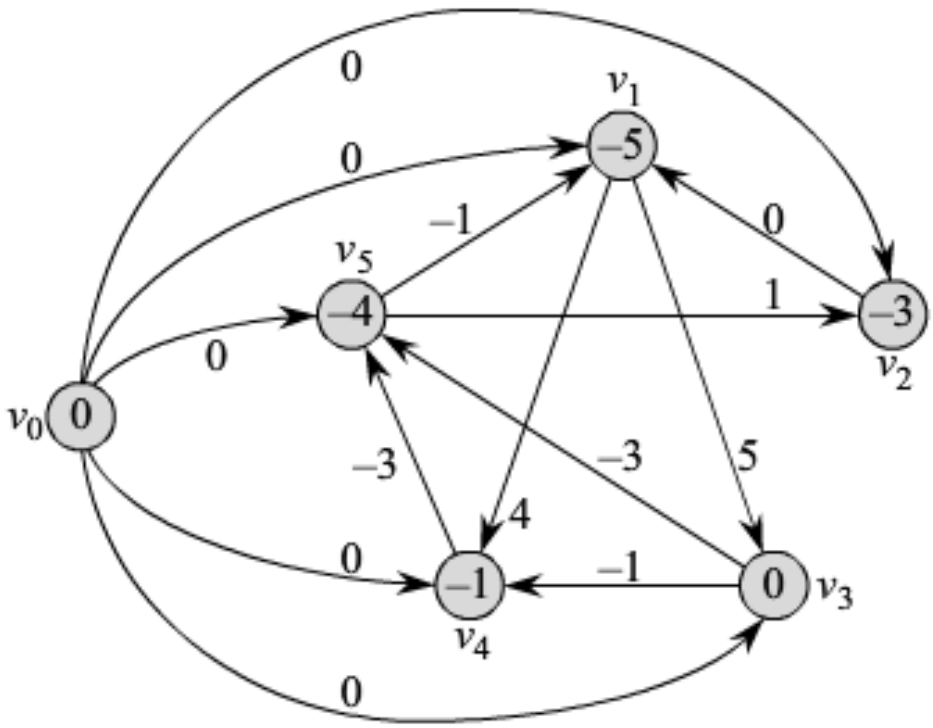
Computing New Weights

- Reweight $\widehat{w}$ using function h(v) for v in V.
- $\widehat{w}$(u, v) = w(u, v) + h(u) – h(v)

- Reweighting doesn't change Shortest Path
- Reweighting maintains negative weight cycles

$$\begin{aligned} \widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) \,, \end{aligned}$$

Example Constraint Graph



- Uses similar graph to produce reweighting function h(v)

Reweighting Function h(v)

- Given: G=(V,E) w/ w : E -> R
- Make G’ = (V’,E’) where
  - V’ = V + {s}
  - E’ = E + {(s, v) : v in V}
- Same as our Difference Constraints Graph

Reweighting Function h(v)

- Given: G=(V,E) w/ w : E -> R
- Make G’ = (V’,E’) where
  - V’ = V + {s}
  - E’ = E + {(s, v) : v in V}
- Now G’:
  - No shortest path include s unless they start with s.
  - No negative weight-cycle unless in G.
- Define h(v) = δ(s,v) for all v in V’

Reweighting Function h(v)

- Given: G=(V,E) w/ w : E -> R
- Make G' = (V',E') where
  - V' = V + {s}
  - E' = E + {(s, v) : v in V}

- Define h(v) = δ(s,v) for all v in V'
  - $h(v) \leq h(u) + w(u,v)$
  - $0 \leq h(u) - h(v) + w(u,v)$
  - $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$
    - NONNEGATIVE

```
JOHNSON(G, w)
compute G′, where G′.V = G.V ∪ {s},
        G′.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
if BELLMAN-FORD(G′, w, s) == FALSE
        print "the input graph contains a negative-weight cycle"
else for each vertex v ∈ G′.V
                set h(v) to the value of δ(s, v)
                        computed by the Bellman-Ford algorithm
        for each edge (u, v) ∈ G′.E
                ŵ(u, v) = w(u, v) + h(u) − h(v)
        let D = (d_uv) be a new n × n matrix
        for each vertex u ∈ G.V
                run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
                for each vertex v ∈ G.V
                        d_uv = δ̂(u, v) + h(v) − h(u)
        return D
```

Algorithm Design

- Divide-and-Conquer
  - Break Problem into Independent Subproblems
- Greedy
  - Break problem into Greedy Choice and One Subproblem.
- Dynamic Programming
  - Break problem into Overlapping Subproblems.

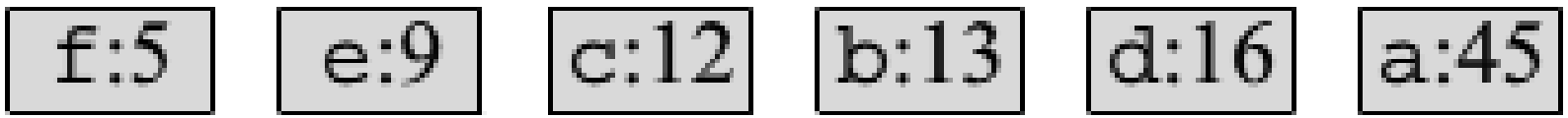
Greedy Algorithm Design

1. Cast the optimization problem as one in which:
   1. We make a choice
   2. Are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice
3. Demonstrate optimal substructure by showing that,
   1. Having made the greedy choice,
   2. What remains is a subproblem where if we combine an optimal solution to the subproblem with our greedy choice we arrive at an optimal solution to the original problem.

Key Properties for Greedy

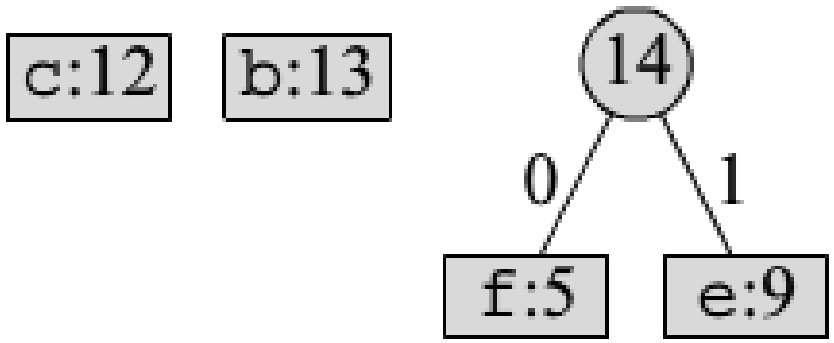
- Greedy-Choice Property:
  - We can assemble a globally optimal solution by making locally optimal (greedy) choices.
- Optimal Substructure:
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Huffman Algorithm Example



- Initial Characters w/ Frequencies

Huffman Algorithm Example



allocate a new node z

$z.left = x = \text{EXTRACT-MIN}(Q)$

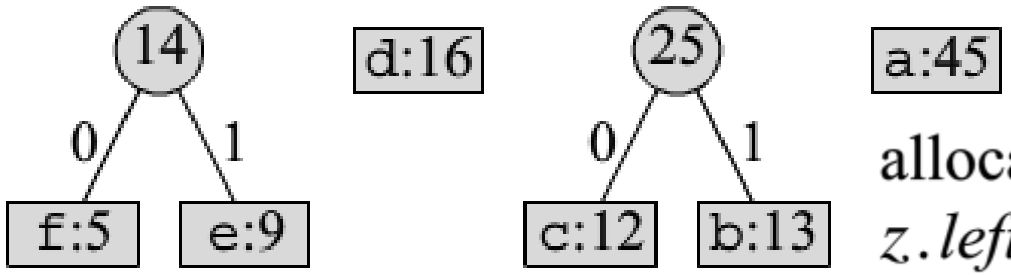
$z.right = y = \text{EXTRACT-MIN}(Q)$

$z.freq = x.freq + y.freq$

$\text{INSERT}(Q, z)$

- First Iteration
  - Allocate Node
  - Extract-Min from Q for Left: [f:5]
  - Extract-Min from Q for Right: [e:9]
  - Sum Frequencies and add New Node to Q
    - [14: [f:5], [e:9]]

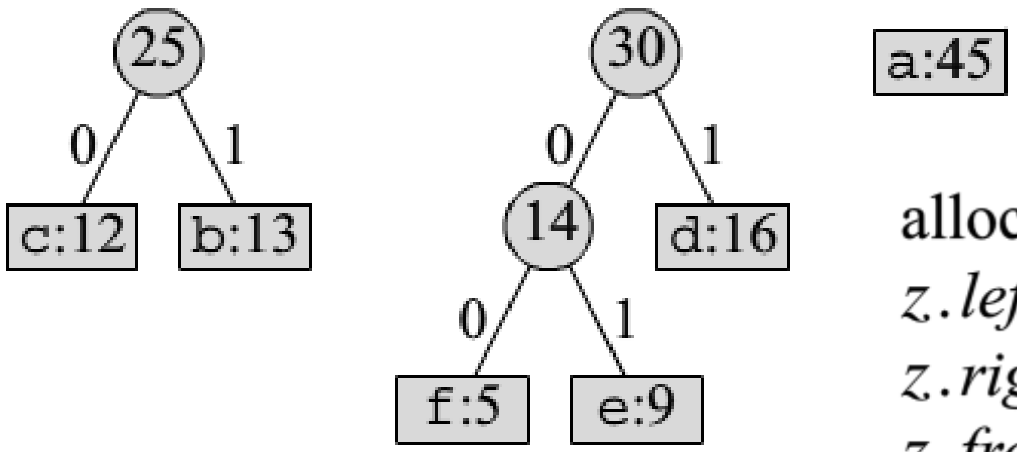
Huffman Algorithm Example



allocate a new node z
$z.left = x = \text{EXTRACT-MIN}(Q)$
$z.right = y = \text{EXTRACT-MIN}(Q)$
$z.freq = x.freq + y.freq$
$\text{INSERT}(Q, z)$

- Second Iteration
  - Allocate Node
  - Extract-Min from Q for Left: [c:12]
  - Extract-Min from Q for Right: [b:13]
  - Sum Frequencies and add New Node to Q
    - [25: [c:12], [b:13]]

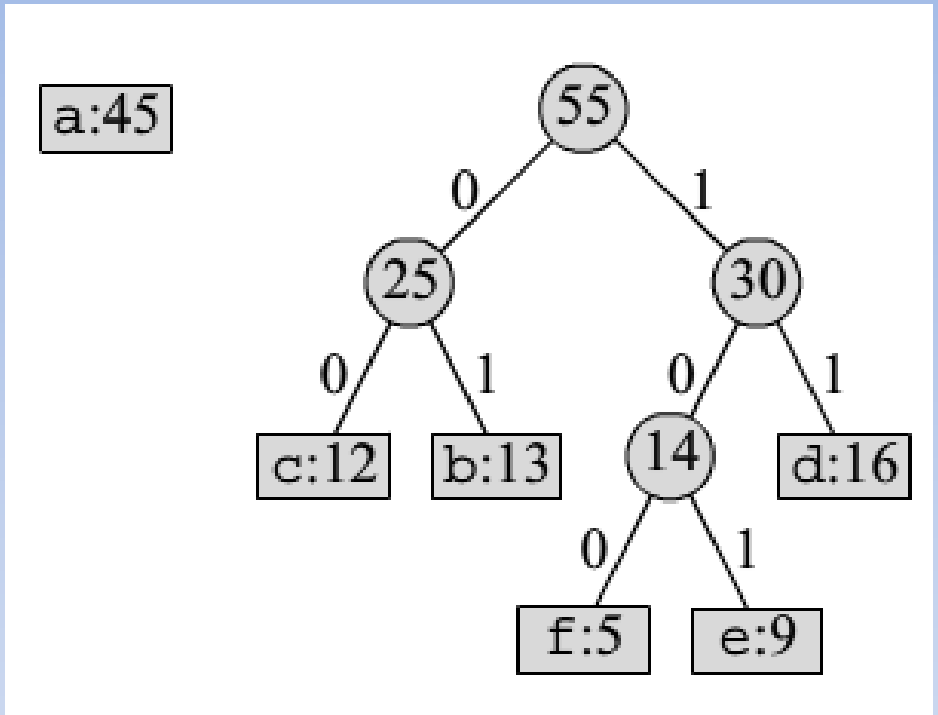
Huffman Algorithm Example



allocate a new node z
$z.left = x = \text{EXTRACT-MIN}(Q)$
$z.right = y = \text{EXTRACT-MIN}(Q)$
$z.freq = x.freq + y.freq$
$\text{INSERT}(Q, z)$

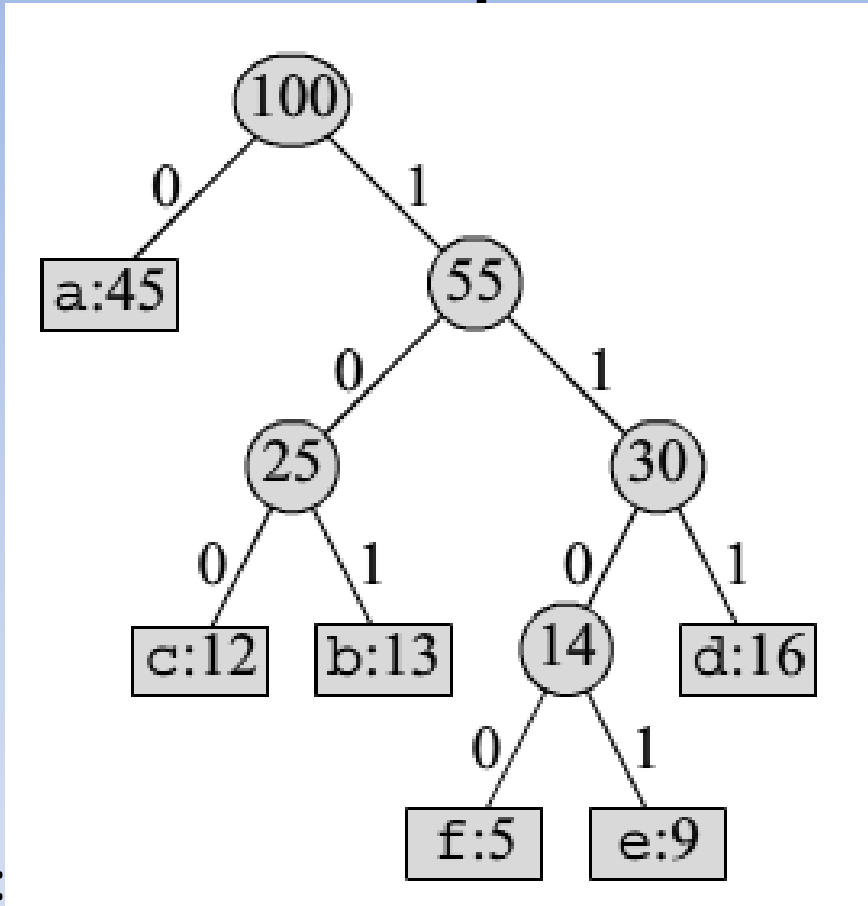
- Third Iteration
  - Allocate Node
  - Extract-Min from Q for Left: [14: [f:5], [e:9]]
  - Extract-Min from Q for Right: [d:16]
  - Sum Frequencies and add New Node to Q
    - [30: [14: [f:5], [e:9]], [d:16]]

Huffman Algorithm Example



- Fourth Iteration
  - Allocate Node
  - Extract-Min from Q for Left:
    - [25: [c:12], [b:13]]
  - Extract-Min from Q for Right:
    - [30: [14: [f:5], [e:9]], [d:16]]
  - Sum Frequencies and add New Node to Q
    - [55: [25: [c:12], [b:13]], [30: [14: [f:5], [e:9]], [d:16]]]

Huffman Algorithm Example



- Final Iteration
  - Allocate Node
  - Extract-Min from Q for Left:
    - [a: 45]
  - Extract-Min from Q for Right:
    - [55: [25: [c:12], [b:13]], [30: [14: [f:5], [e:9]], [d:16]]]
  - Sum Frequencies and add New Node to Q
    - [100: [a:45], [55: [25: [c:12], [b:13]], [30: [14: [f:5], [e:9]], [d:16]]]]

Huffman Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Dynamic Programming

- Subproblems Share Subproblems
- Dynamic Programming Algorithm:
  - solves each subproblem once
  - Save subproblem results in table for re-use.

Designing Dynamic Programming Algorithm

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.
   - Step 4 not needed if only computing optimum value.

Sequence alignment

- Compare two strings to see if they are similar
  - We need to define similarity
  - Very useful in many applications
  - Comparing DNA sequences, articles, source code, etc.

  - Example: Longest Common Subsequence problem (LCS)

Common subsequence

- A subsequence of a string is the string with zero or more chars left out
- A common subsequence of two strings:
  - A subsequence of both strings
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both common subsequences of x and y

Longest Common Subsequence

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x:	A	B	C	B	D	A	B
y:	B	D	C	A	B	A	

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

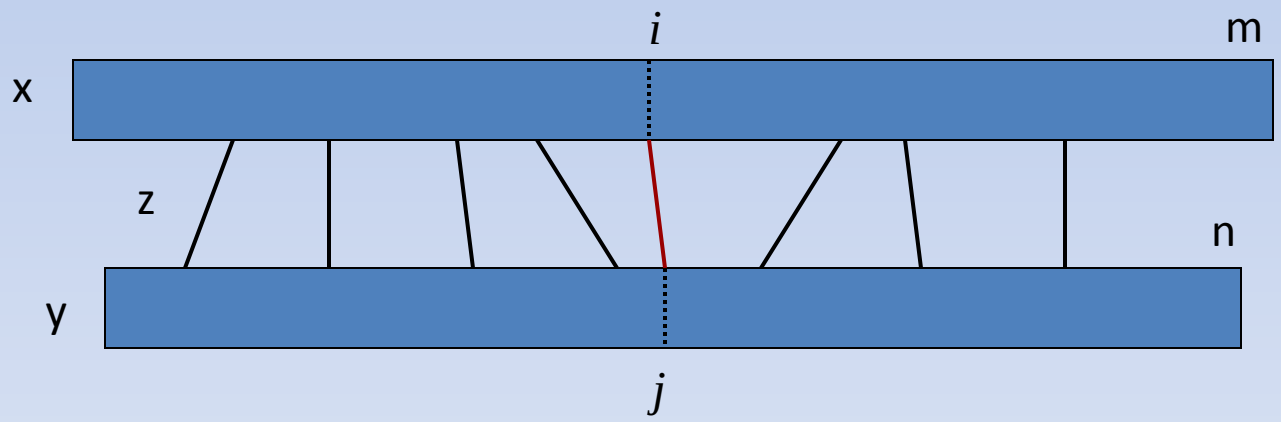
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm: a DP strategy

- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

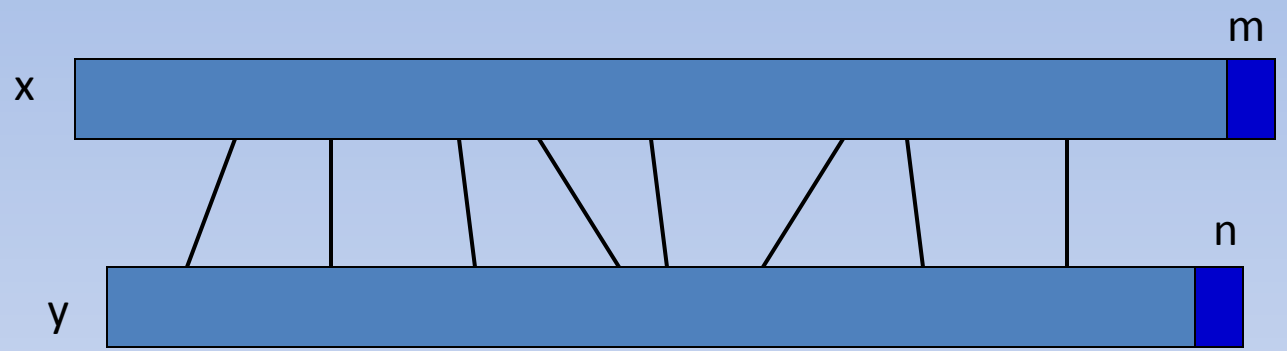
Optimal substructure

- Notice that the LCS problem has *optimal substructure*: parts of the final solution are solutions of subproblems.
  - If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.



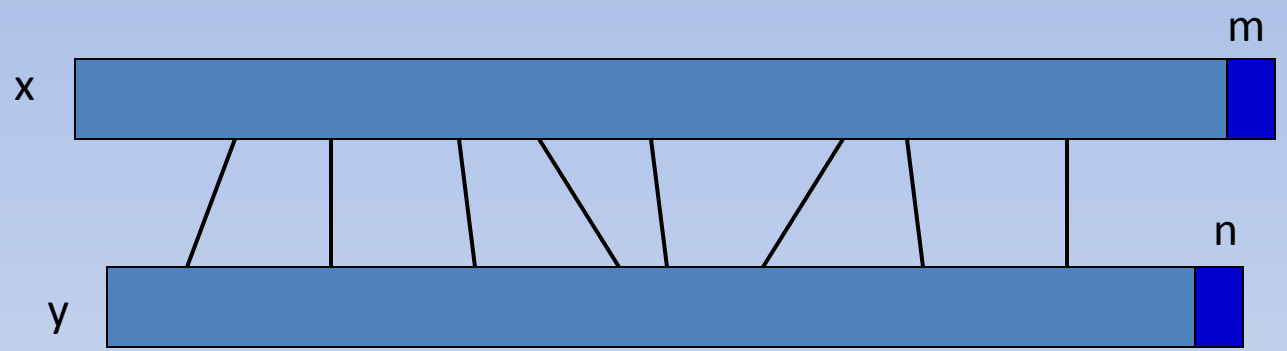
- Subproblems: "find LCS of pairs of *prefixes* of x and y"

Recursive thinking



- Case 1: $x[m]=y[n]$. There is **an** optimal LCS that matches $x[m]$ with $y[n]$. → Find out LCS $(x[1..m-1], y[1..n-1])$
- Case 2: $x[m] \neq y[n]$. At most one of them is in LCS
  - Case 2.1: $x[m]$ *not in LCS* → Find out LCS $(x[1..m-1], y[1..n])$
  - Case 2.2: $y[n]$ *not in LCS* → Find out LCS $(x[1..m], y[1..n-1])$

Recursive thinking



- Case 1: $x[m]=y[n]$

  Reduce both sequences by 1 char

  – $LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) \,||\, x[m]$

  concatenate

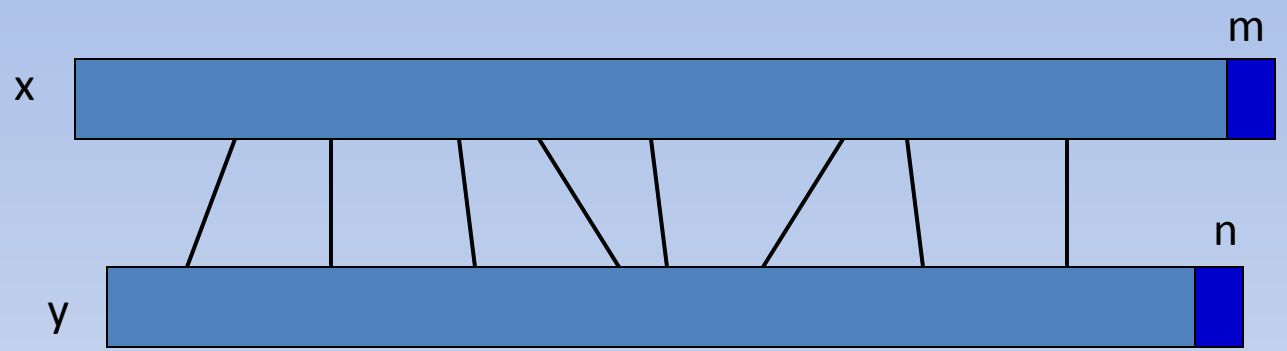
- Case 2: $x[m] \neq y[n]$

  – $LCS(x, y) = LCS(x[1..m-1], y[1..n])$ or

  $LCS(x[1..m], y[1..n-1])$, whichever is longer

  Reduce either sequence by 1 char

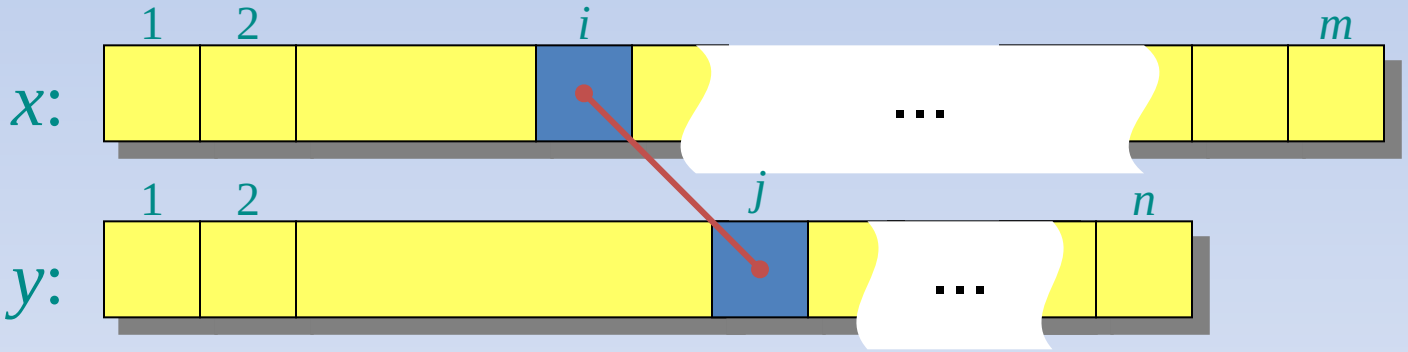
Finding length of LCS



- Let *c[i, j]* be the length of LCS(*x[1..i], y[1..j]*)
  => *c[m, n]* is the length of LCS(*x, y*)
- If $x[m] = y[n]$
  $c[m, n] = c[m-1, n-1] + 1$
- If $x[m] \mathrel{!}= y[n]$
  $c[m, n] = max\{ c[m-1, n], c[m, n-1] \}$

Generalize: recursive formulation

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

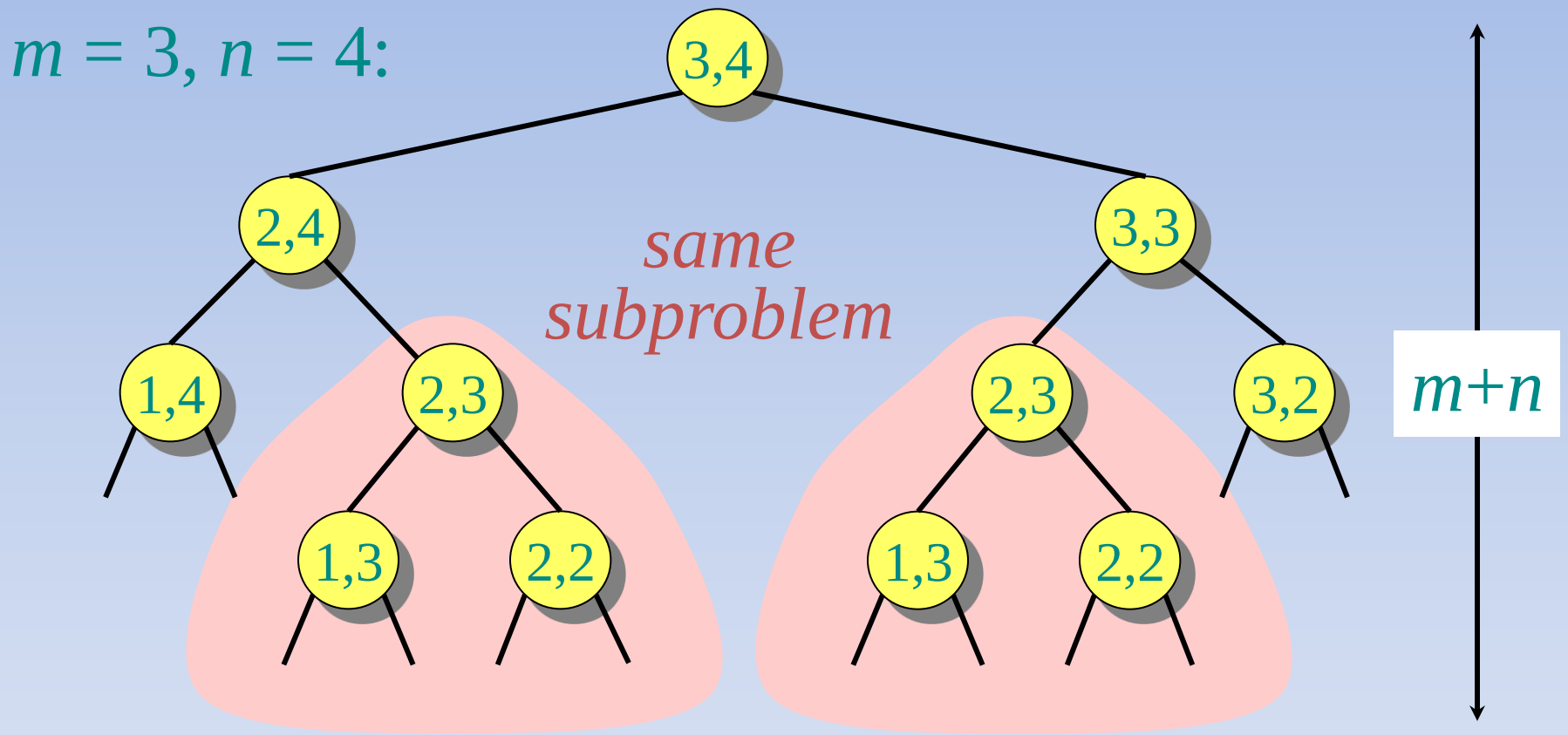


Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$,
      LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

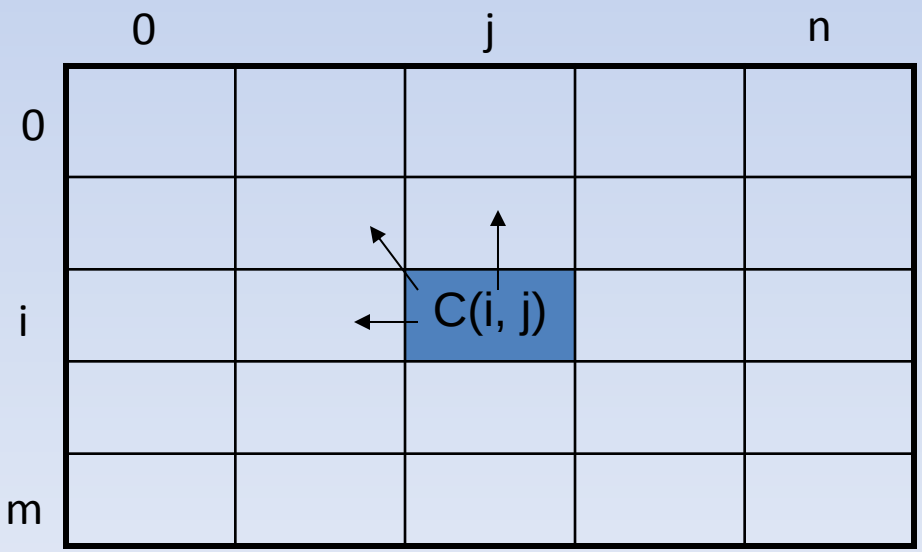


Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: *m * n*

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$



DP Algorithm

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n  = length(Y) // get the # of symbols in Y
for i = 1 to m  c[i,0] = 0   // special case: Y[0]
for j = 1 to n   c[0,j] = 0   // special case: X[0]
for i = 1 to m                // for all X[i]
    for j = 1 to n                // for all Y[j]
        if ( X[i] == Y[j])
            c[i,j] = c[i-1,j-1] + 1
        else c[i,j] = max( c[i-1,j], c[i,j-1] )
return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
for i = 1 to m        c[i,0] = 0
for j = 1 to n        c[0,j] = 0
```

LCS Example (2)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (9)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**				

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if (X_i == Y_j)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	3

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was acquired:

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Time for trace back: $O(m+n)$.

Finding LCS (2)

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

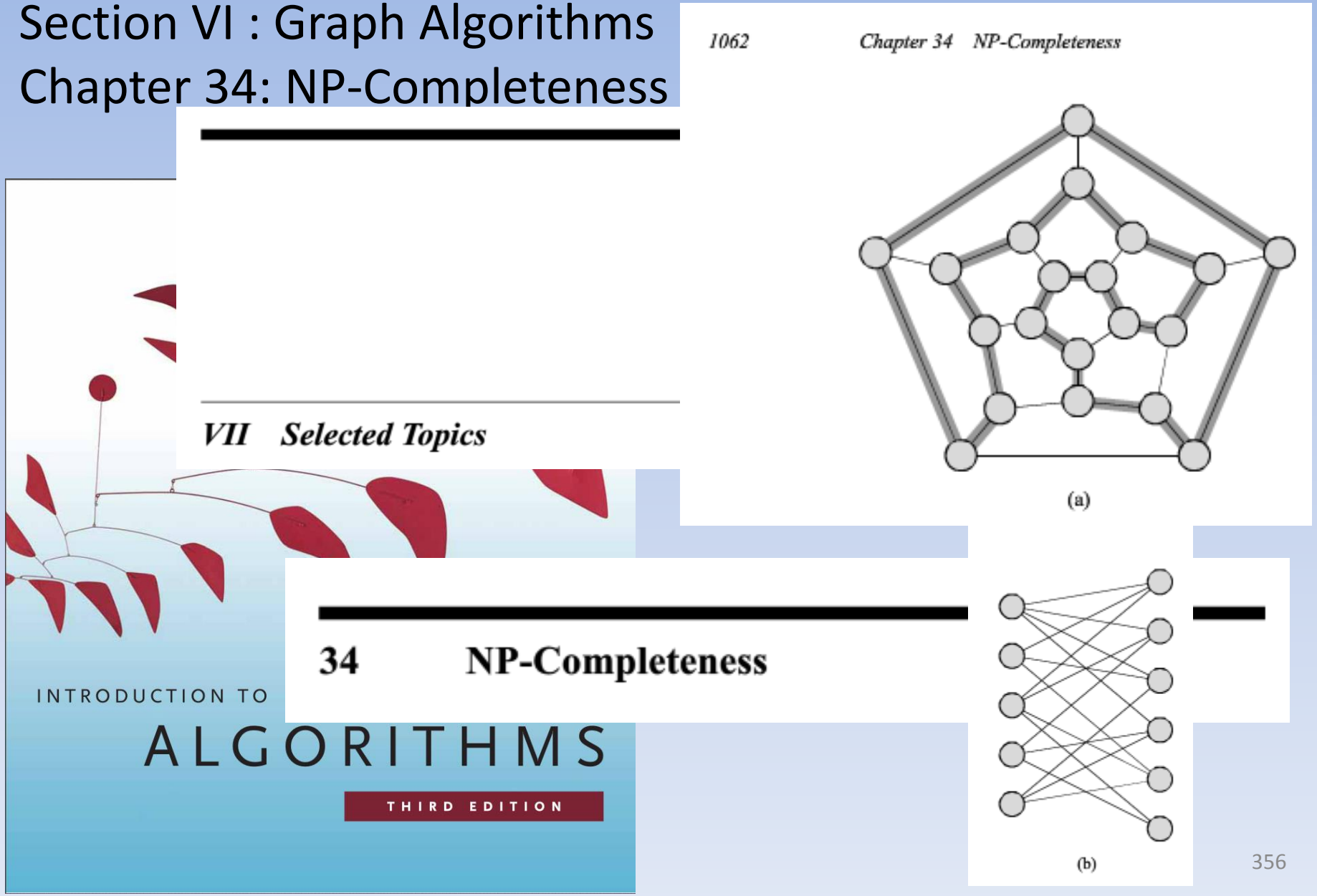
LCS (straight order): **B C B**

(this string turned out to be a palindrome)

Design and Analysis of Algorithms

Section VI : Graph Algorithms

Chapter 34: NP-Completeness

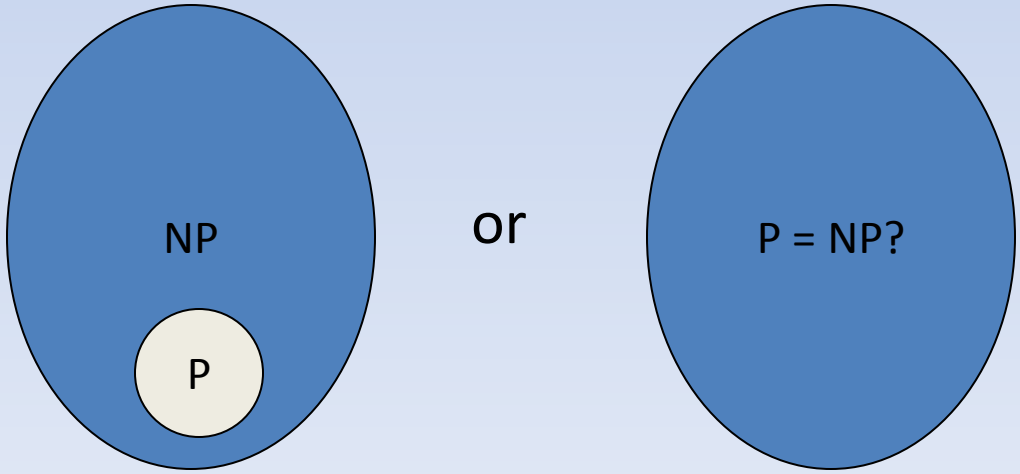


P and NP

- **P** = problems that can be solved in polynomial time
- **NP** = problems for which a solution can be verified in polynomial time
- K-clique problem is in **NP**:
  - Easy to verify solution in polynomial time
- Is sorting in NP?
  - Not a decision problem
- Is sortedness in NP?
  - Yes. Easy to verify

P and NP

- Is **P = NP?**
  - The biggest open problem in CS
  - Most suspect not
  - the Clay Mathematics Institute has offered a $1 million prize for the first proof



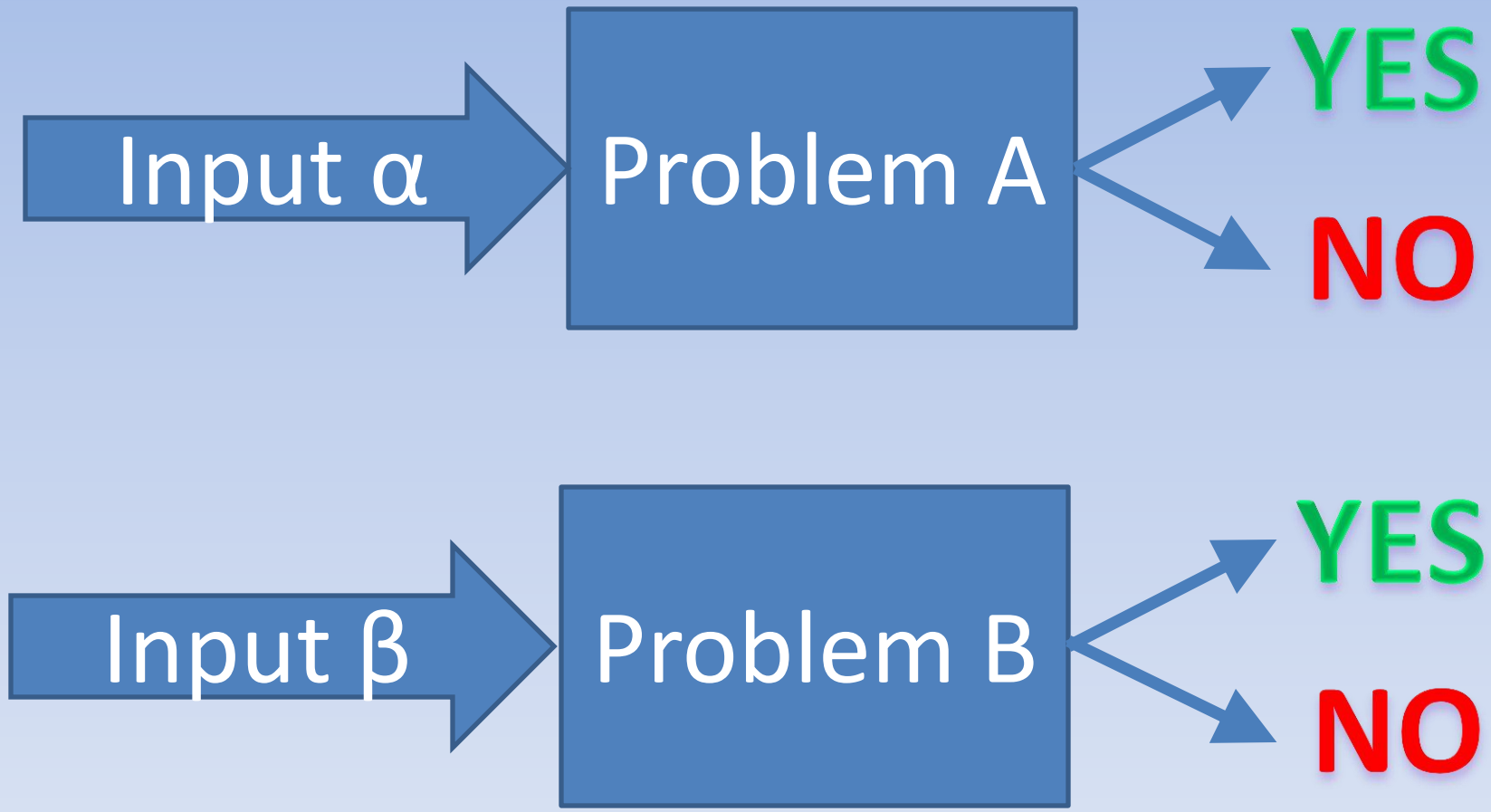
NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve TSP in $O(n^{100})$ time, you've proved that **P = NP**. Retire rich & famous.

Reduction

- The crux of NP-Completeness is *reducibility*
  - Informally, a problem Π' can be reduced to another problem Π if *any* instance of Π' can be "easily rephrased" as an instance of Π, the solution to which provides a solution to the instance of Π'
    - *What do you suppose "easily" means?*
    - This rephrasing is called *transformation*
  - Intuitively: If Π' reduces to Π, Π' is "no harder to solve" than Π

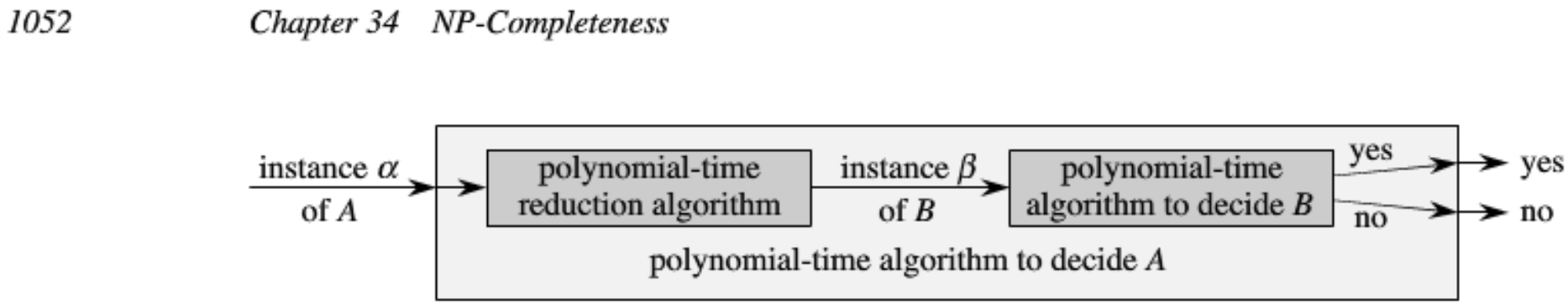
Reductions w/ Two Different Problems



Reductions w/ Transformations

- Introduce a polynomial time algorithm to transform an instance α into an instance β.
- The answer for instance α (YES | NO) is the same as β (YES | NO).
  - If the answer for α is YES, then the answer for β is YES.

Reductions w/ Polynomial-Time Reduction Algorithm



- Assume we have a polynomial time algorithm for problem B.
- Assume we have a polynomial time algorithm to transform an instance α into an instance β.
- Construct polynomial-time Reduction Algorithm:
  - Take input α convert in polynomial time to β
  - Run polynomial-time decision algorithm for B on β
  - Use answer for β as the answer for α

Polynomial-Time Reduction Algorithm

- It is easy to see that given:
  - Polynomial-Time algorithm for converting α to β
  - Polynomial-Time algorithm for answering β
  - Able to utilize answer to β for α
- We'll have a polynomial-time algorithm for answering α!
- Therefore: we can assert that A is as easy as B!

Reductions for Hardness

- Our goal is not to show how easy a problem is, but to define a class of HARD problems.
- We do this by defining the class of NP-Complete problems, as hard as any other problem in NP.
- We then use Reductions to show a problem is NP-Complete.

Reductions to Intractable Problems

- Suppose we know that problem A is intractable:
  - Provably problem has no polynomial-time solution.
    - Exhaustive search in an exponential space!
- Assume we have a polynomial-time reduction from a second problem B into this provably intractable problem.
- Now with a proof by contradiction we can prove that B is intractable!!
  - If we had a polynomial-time algorithm for B we would have one for A.
  - But there can be no polynomial-time algorithm for A by our initial assumption!!

Reductions and NP-Completeness

- For NP-Complete problems we do not know for sure that these problems are intractable.
- For NP-Complete problems we do know that no polynomial-time algorithm has ever been found for any of them.
- For NP-Complete problems we can use reductions to prove that a polynomial-algorithm for any of them could immediately produce one for ALL!
- Reduction Technique requires an existing NP-Complete problem
  - Book Introduces Circuit-Satisfiability

Problems and Polynomial Time

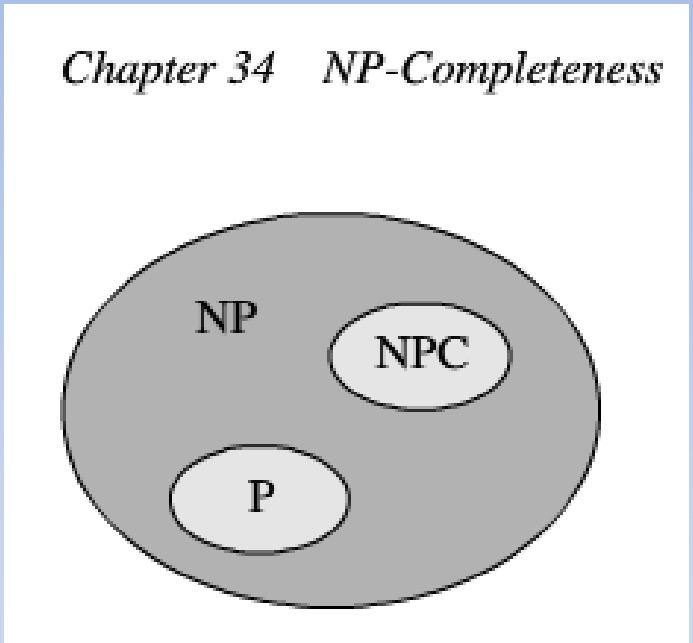
- Note that high order polynomial-time algorithm aren't really tractable.
  - We've already looked at n^2 versus $n \lg n$
- Why should we consider class of problems order some polynomial as tractable?

- Answer is more Philosophical!

Problems and Polynomial Time

1. Problems with polynomial-time solutions tend towards lower order polynomials.
   - Once polynomial time solutions found, improved versions tend to be found.
2. Polynomial-time algorithms remain polynomial time under multiple models of computation.
   - Random-Access Machine
   - Turing Machines
   - Parallel computer where processors grows polynomial w/ input size.
3. Polynomial-time algorithms are nicely closed under a variety additive of multiplicative operations.

Current View on P & NP & NPC



- Both P and NPC are wholly contained with NP
- $P \cap NPC = \emptyset$