

# Discrete Mathematics for Computer Science

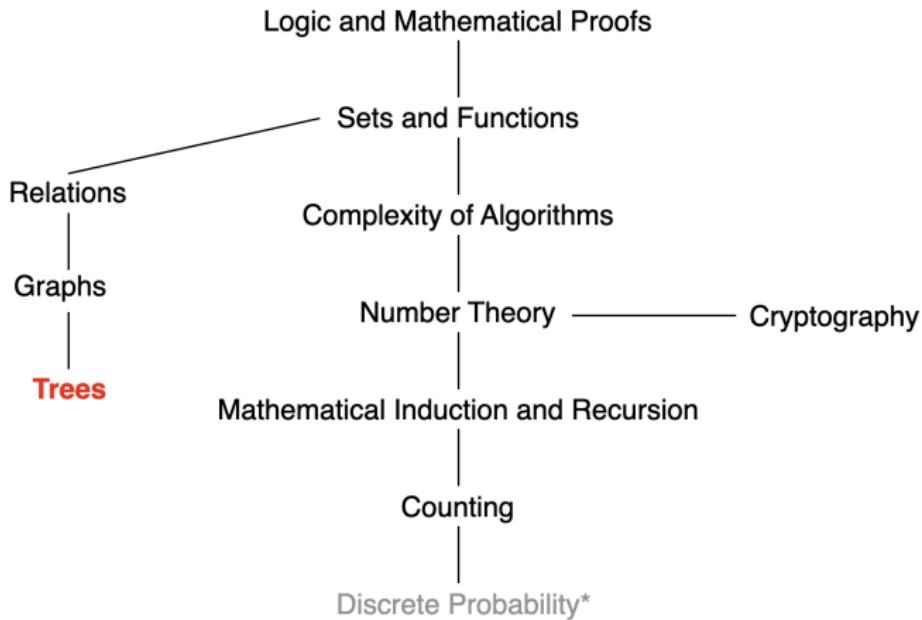
## Lecture 20: Graph

Dr. Ming Tang

Department of Computer Science and Engineering  
Southern University of Science and Technology (SUSTech)  
Email: tangm3@sustech.edu.cn



# This Lecture

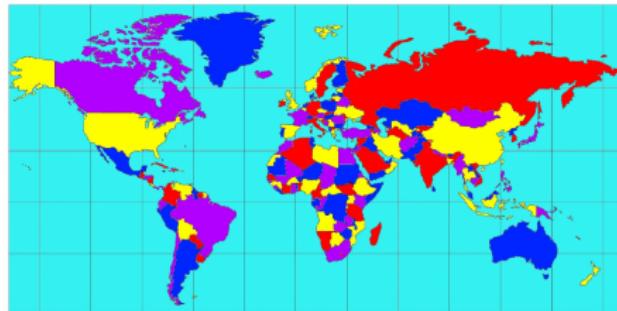


..., Euler and Hamilton path, shortest-path problem, planar graphs, graph coloring, ...

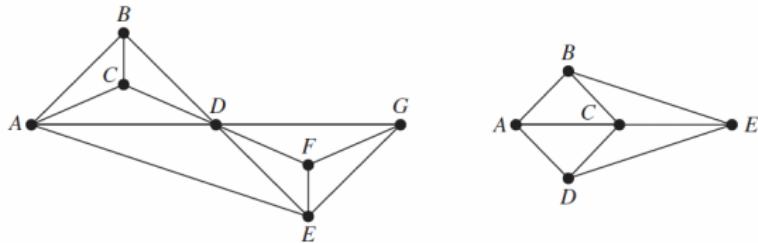
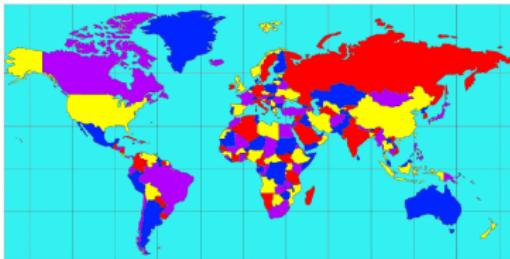


# Graph Coloring

**Four-color theorem:** Given any separation of a plane into contiguous regions, producing a figure called a map, **no more than four colors** are required to color the regions of the map so that no two adjacent regions have the same color.



# Graph Coloring



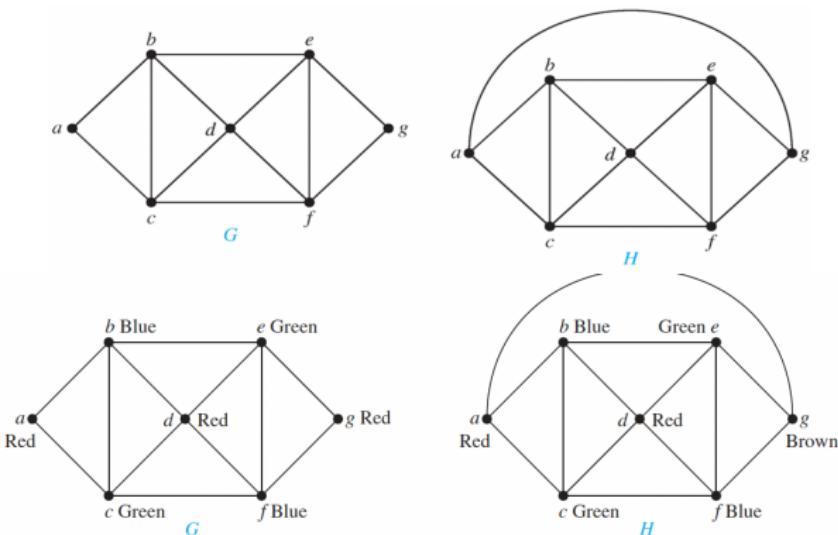
- A map can be represented as a planar graph
- A **coloring** of a simple graph is the assignment of a color to each vertex of the graph so that **no two adjacent vertices** are assigned the same color.

# Graph Coloring

The **chromatic number** of a graph is the least number of colors needed for a coloring of this graph, denoted by  $\chi(G)$ .

**Theorem** (Four Color Theorem): The **chromatic number** of a planar graph is **no greater than four**.

# Graph Coloring: Example

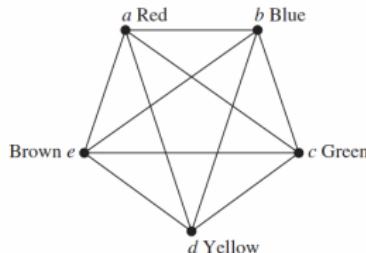


The chromatic number of  $G$  is at least three, because the vertices  $a$ ,  $b$ , and  $c$  must be assigned different colors.

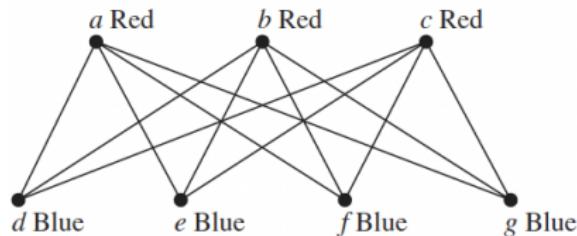
- Assign red to  $a$ , blue to  $b$ , and green to  $c$ ;
- $d$  can (and must) be colored red because it is adjacent to  $b$  and  $c$ ;
- $e$  can (and must) be colored green because it is adjacent only to vertices colored red and blue;

## Graph Coloring: Example

What is the chromatic number of  $K_n$ ?  $\chi(K_n) = n$

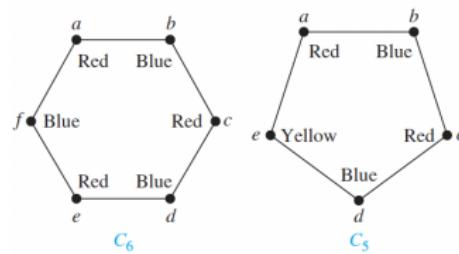


What is the chromatic number of the complete bipartite graph  $K_{m,n}$ , where  $m$  and  $n$  are positive integers?  $\chi(K_{m,n}) = 2$



# Graph Coloring: Example

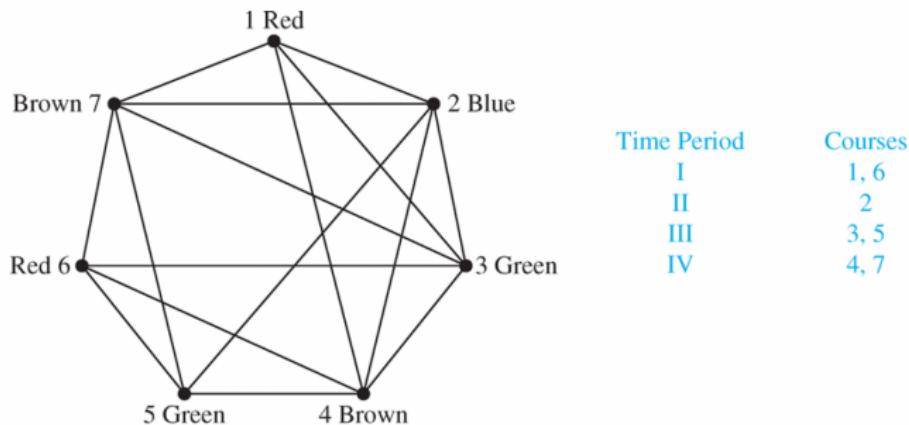
What is the chromatic number of the graph  $C_n$ , where  $n \geq 3$ ? (Recall that  $C_n$  is the cycle with  $n$  vertices.)



- When  $n$  is even,  $\chi(C_n) = 2$
- When  $n$  is odd and  $n > 1$ ,  $\chi(C_n) = 3$

# Applications of Graph Coloring

Scheduling Final Exams: Vertices represent courses, and there is an edge between two vertices if there is a common student in the courses.



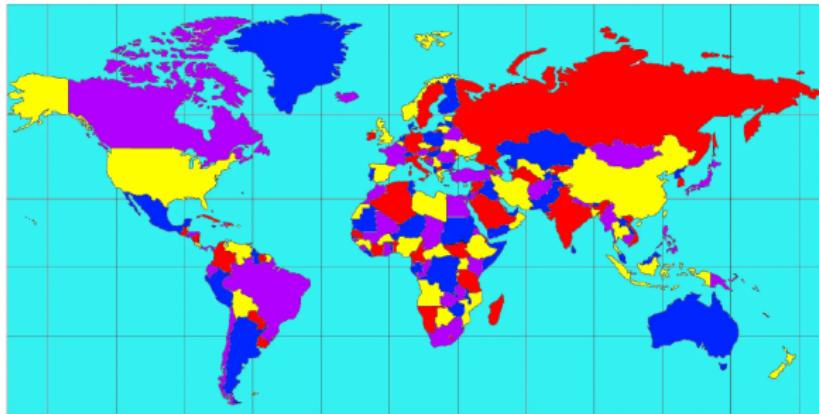
# Applications of Graph Coloring

**Channel Assignments:** Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel. How can the assignment of channels be modeled by graph coloring?

Graph Coloring  $\in$  NP-Complete

# Graph Coloring

**Theorem (Four Color Theorem):** The chromatic number of a planar graph is no greater than four.



# Graph Coloring

**Theorem (Six Color Theorem):** The chromatic number of a planar graph is no greater than six.

**Proof** (by induction on the number of vertices): W.l.o.g., assume that the graph is connected.

- **Basic step:** For one single vertex, pick an arbitrary color.
- **Inductive hypothesis:** Assume that every planar graph with  $k \geq 1$  or fewer vertices can be **6-colored**.
- **Inductive step:** Consider a planar graph with  $k + 1$  vertices. Recall Corollary 2 (the graph has a vertex of degree 5 or fewer). Remove this vertex, by i.h., we can color the remaining graph with 6 colors. Put the vertex back in. Since there are at most 5 colors adjacent, so we have at least one color left.

# Graph Coloring

**Theorem (Five Color Theorem):** The chromatic number of a planar graph is no greater than five.

**Proof** (by induction on the number of vertices): W.l.o.g., assume that the graph is connected.

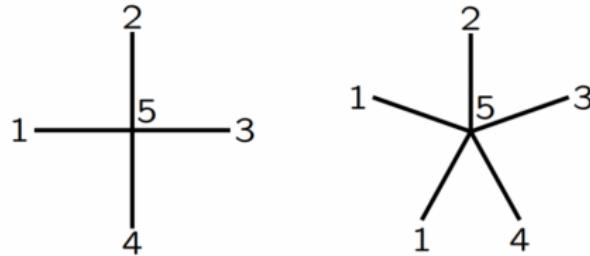
- **Basic step:** For one single vertex, pick an arbitrary color.
- **Inductive hypothesis:** Assume that every planar graph with  $k \geq 1$  or fewer vertices can be **5-colored**.
- **Inductive step:** Consider a planar graph with  $k + 1$  vertices. Recall Corollary 2 (the graph has a vertex of degree 5 or fewer). Remove this vertex, by i.h., we can color the remaining graph with 6 colors. Put the vertex back in.

## Graph Coloring

**Theorem (Five Color Theorem):** The chromatic number of a planar graph is no greater than five.

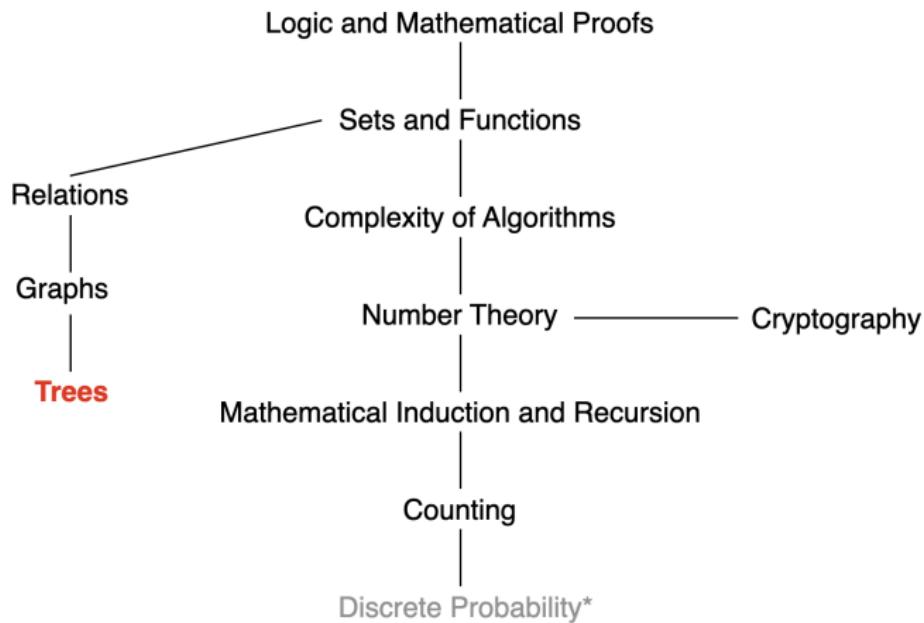
**Proof** (by induction on the number of vertices): W.l.o.g., assume that the graph is connected.

**Case 1:** If the vertex has degree less than 5, or if it has degree 5 and only  $\leq 4$  colors are used for vertices connected to it, we can pick an available color for it.



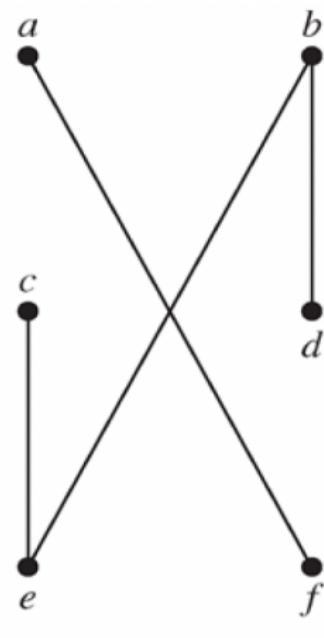
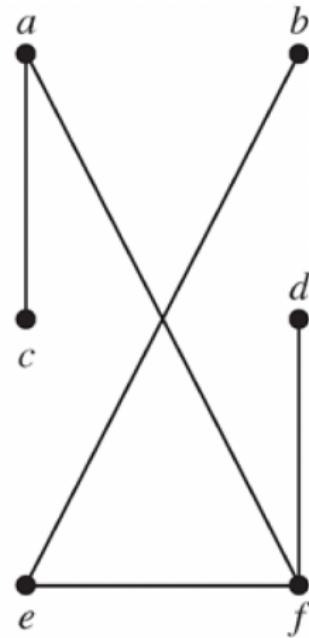
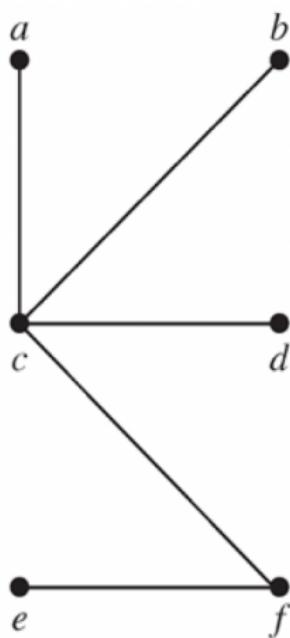
**Case 2:** If the vertex has degree 5, and all 5 colors are connected to it, we label the vertices adjacent to the “special” vertex (degree 5) in order).

# This Lecture



# Trees

**Definition:** A **tree** is a connected undirected graph with **no simple circuits**.



Nikolaus  
(1623–1708)

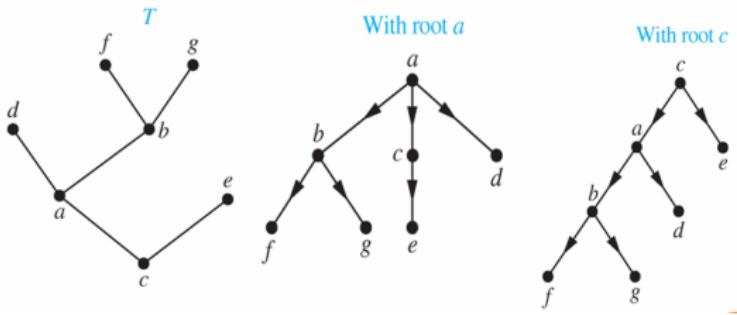
# Trees

**Theorem:** An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

**Proof:** Two properties of tree: connected, no circuit

# Rooted Trees

**Definition:** A **rooted tree** is a tree in which one vertex has been designated as the root and every edge is directed away from the root.



We can change an unrooted tree into a rooted tree by choosing any vertex as the root.

The arrows indicating the directions of the edges in a rooted tree can be omitted, because the choice of root determines the directions of the edges.

# Rooted Trees

The **parent** of  $v$  is the unique vertex  $u$  such that there is a directed edge from  $u$  to  $v$ .

When  $u$  is the parent of  $v$ ,  $v$  is called a **child** of  $u$ .

Vertices with the same parent are called **siblings**.

The **ancestors** of a vertex are the vertices in the path from the root to this vertex, **excluding** the vertex itself and **including** the root.

The **descendants** of a vertex  $v$  are those vertices that have  $v$  as an ancestor.

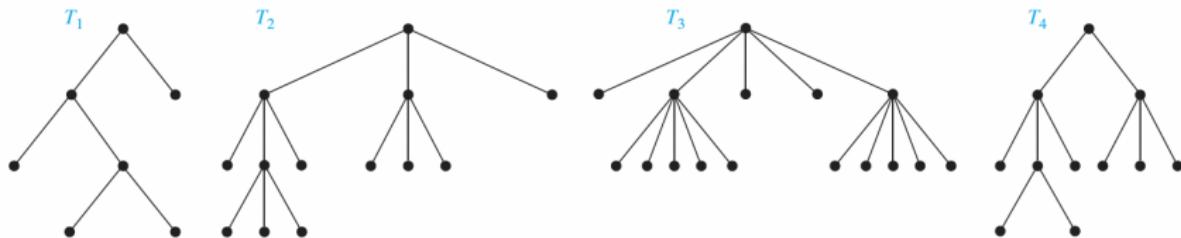
A vertex of a rooted tree is called a **leaf** if it has no children.

Vertices that have children are called **internal vertices**.

**Subtree with  $a$  as its root:** consists of  $a$  and its descendants and all edges incident to these descendants.

# $m$ -Ary Trees

**Definition:** A rooted tree is called an  *$m$ -ary tree* if every internal vertex has no more than  $m$  children. The tree is called a full  *$m$ -ary tree* if every internal vertex has exactly  $m$  children. In particular, an  $m$ -ary tree with  $m = 2$  is called a *binary tree*.



# Ordered Rooted Tree

**Definition:** An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order **from left to right**.

In an **ordered binary tree** (usually called just a binary tree), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**.

The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex.

# Counting Vertices in a Full $m$ -Ary Trees

**Theorem:** A tree with  $n$  vertices has  $n - 1$  edges.

**Theorem:** A full  $m$ -ary tree with  $i$  internal vertices has  $n = mi + 1$  vertices.

**Theorem:** A full  $m$ -ary tree with

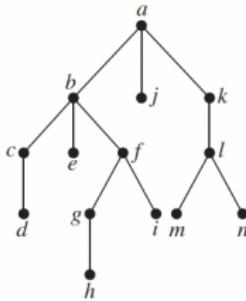
- (i)  $n$  vertices has  $i = (n - 1)/m$  internal vertices and  $l = [(m - 1)n + 1]/m$  leaves,
- (ii)  $i$  internal vertices has  $n = mi + 1$  vertices and  $l = (m - 1)i + 1$  leaves,
- (iii)  $l$  leaves has  $n = (ml - 1)/(m - 1)$  vertices and  $i = (l - 1)/(m - 1)$  internal vertices.

Using  $n = mi + 1$  and  $n = i + l$

# Level and Height

The **level** of a vertex  $v$  in a rooted tree is the length of the unique path from the root to this vertex.

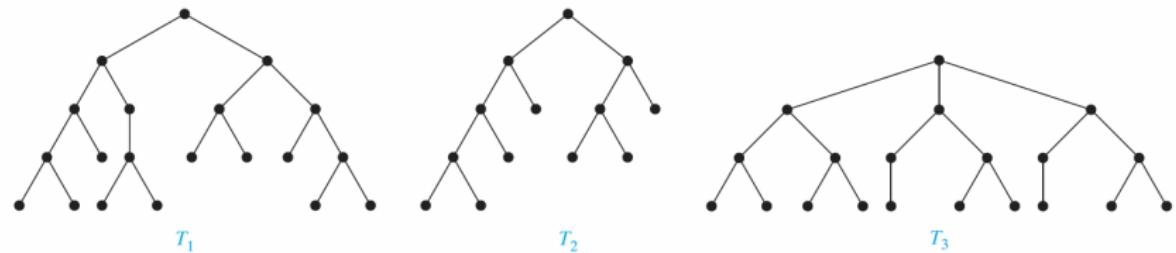
The **height** of a rooted tree is the maximum of the levels of the vertices.



The root  $a$  is at level 0. Vertices  $b$ ,  $j$ , and  $k$  are at level 1. ... Finally, vertex  $h$  is at level 4. Because the largest level of any vertex is 4, this tree has height 4.

# Level and Height

A rooted  $m$ -ary tree of height  $h$  is balanced if all leaves are at levels  $h$  or  $h - 1$ .



$T_1$  is balanced, because all its leaves are at levels 3 and 4.

$T_2$  is not balanced, because it has leaves at levels 2, 3, and 4.

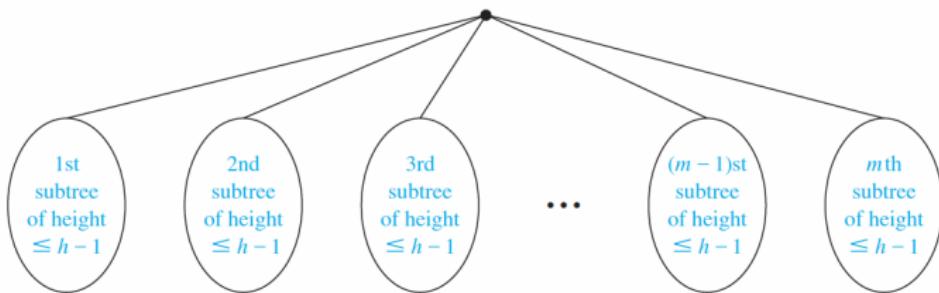
$T_3$  is balanced, because all its leaves are at level 3.

# The Number of Leaves

**Theorem:** There are at most  $m^h$  leaves in an  $m$ -ary tree of height  $h$ .

**Proof** (by induction)

- Basic Step:  $h = 1$ , at most  $m$  leaves
- Inductive Step: Now assume that the result is true for all  $m$ -ary trees of height less than  $h$ ; Consider an  $m$ -ary tree of height  $h$ :



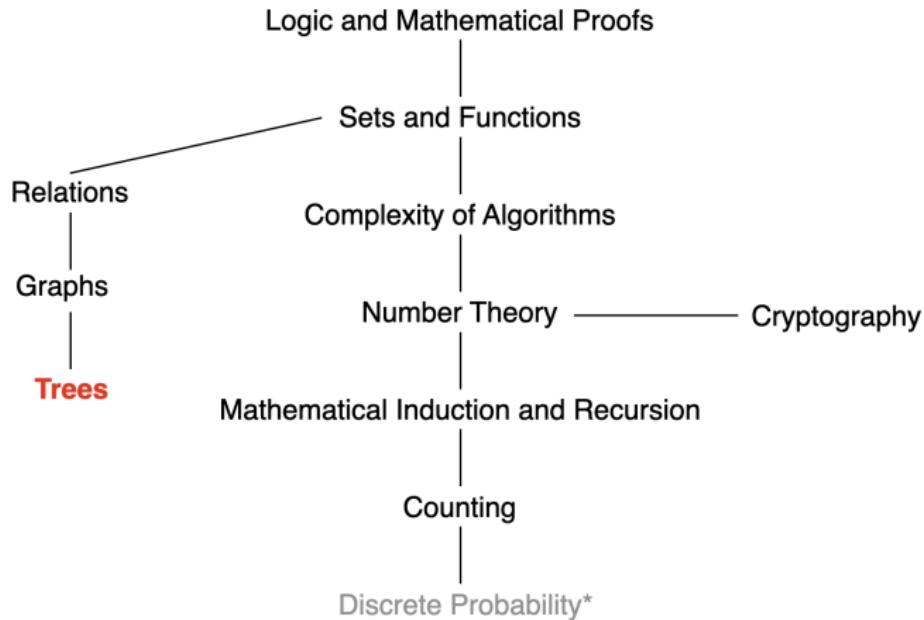
# The Number of Leave

**Corollary:** If an  $m$ -ary tree of height  $h$  has  $l$  leaves, then  $h \geq \lceil \log_m l \rceil$ . If the  $m$ -ary tree is **full** and **balanced**, then  $h = \lceil \log_m l \rceil$ .

**Proof:**

- We know that  $l \leq m^h$ . Taking logarithms to the base  $m$  shows that  $\log_m l \leq h$ . Because  $h$  is an integer, we have  $h \geq \lceil \log_m l \rceil$ .
- Now suppose that the tree is balanced. Then, each leaf is at level  $h$  or  $h - 1$ , and because the height is  $h$ , there is at least one leaf at level  $h$ . Since the tree is full, there must be more than  $m^{h-1}$  leaves.
- Because  $l \leq m^h$ , we have  $m^{h-1} < l \leq m^h$ . Taking logarithms to the base  $m$  in this inequality gives  $h - 1 < \log_m l \leq h$ . Hence,  $h = \lceil \log_m l \rceil$ .

# This Lecture



Tree, Tree Traversal, ...

# Tree Traversal

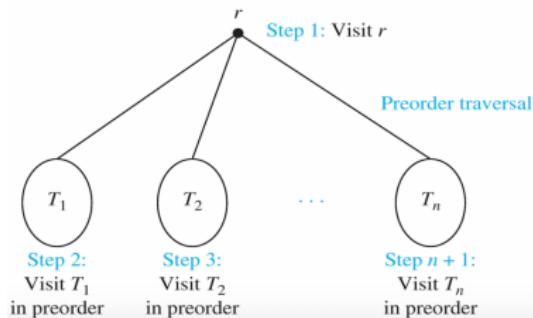
The procedures for systematically visiting every vertex of an ordered tree are called traversals.

The three most commonly used traversals are preorder traversal, inorder traversal, postorder traversal.

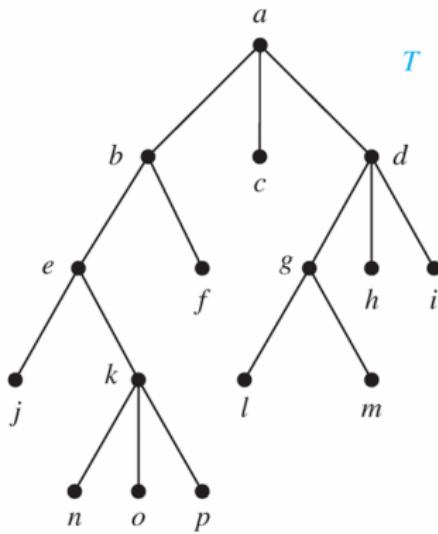
# Preorder Traversal

**Definition** Let  $T$  be an ordered rooted tree with root  $r$ . If  $T$  consists only of  $r$ , then  $r$  is the **preorder traversal** of  $T$ .

Otherwise, suppose that  $T_1, T_2, \dots, T_n$  are the subtrees of  $r$  from left to right in  $T$ . The preorder traversal **begins by visiting  $r$** , and continues by traversing  $T_1$  **in preorder**, then  $T_2$  in preorder, and so on, until  $T_n$  is traversed in preorder.



# Preorder Traversal: Example



a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i

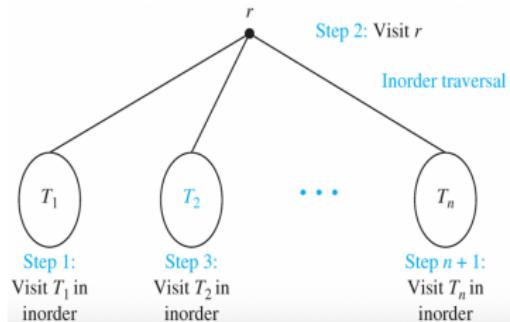
# Preorder Traversal: Algorithm

```
procedure preorder ( $T$ : ordered rooted tree)
     $r :=$  root of  $T$ 
    list  $r$ 
    for each child  $c$  of  $r$  from left to right
         $T(c) :=$  subtree with  $c$  as root
        preorder( $T(c)$ )
```

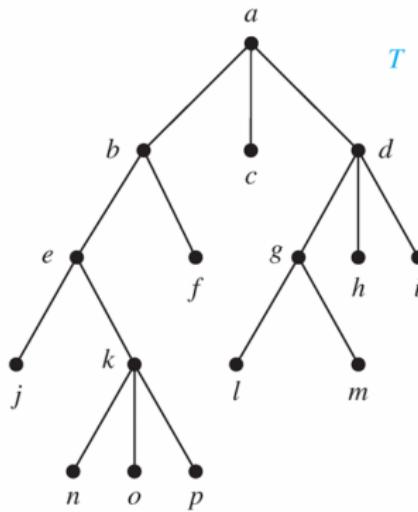
# Inorder Traversal

**Definition:** Let  $T$  be an ordered rooted tree with root  $r$ . If  $T$  consists only of  $r$ , then  $r$  is the inorder traversal of  $T$ .

Otherwise, suppose that  $T_1, T_2, \dots, T_n$  are the subtrees of  $r$  from left to right in  $T$ . The inorder traversal begins by traversing  $T_1$  in inorder, then visiting  $r$ , and continues by traversing  $T_2$  in inorder, and so on, until  $T_n$  is traversed in inorder.



# Inorder Traversal: Example



j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i

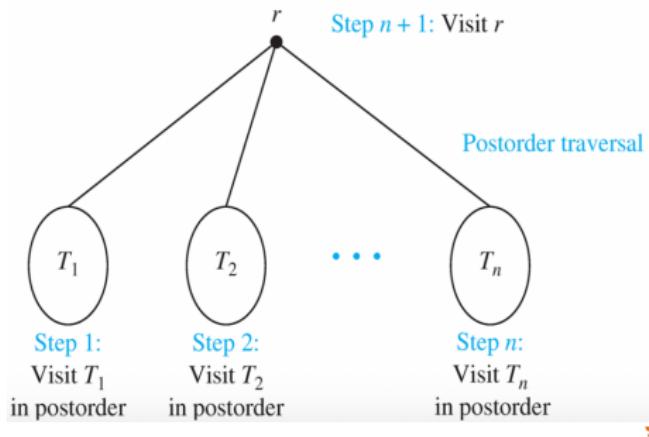
# Inorder Traversal: Algorithm

```
procedure inorder ( $T$ : ordered rooted tree)
 $r :=$  root of  $T$ 
if  $r$  is a leaf then list  $r$ 
else
     $l :=$  first child of  $r$  from left to right
     $T(l) :=$  subtree with  $l$  as its root
    inorder( $T(l)$ )
    list( $r$ )
    for each child  $c$  of  $r$  from left to right
         $T(c) :=$  subtree with  $c$  as root
        inorder( $T(c)$ )
```

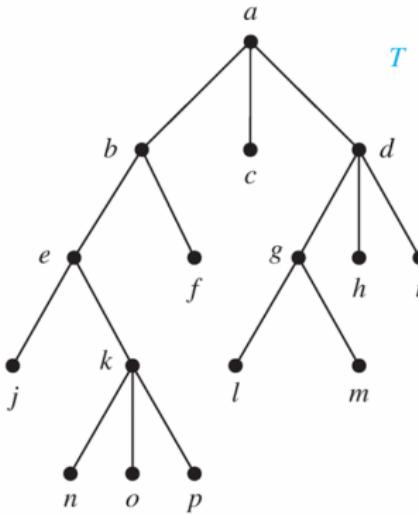
# Postorder Traversal

**Definition:** Let  $T$  be an ordered rooted tree with root  $r$ . If  $T$  consists only of  $r$ , then  $r$  is the postorder traversal of  $T$ .

Otherwise, suppose that  $T_1, T_2, \dots, T_n$  are the subtrees of  $r$  from left to right in  $T$ . The postorder traversal begins by traversing  $T_1$  in postorder, then  $T_2$  in postorder, and so on, after  $T_n$  is traversed in postorder,  $r$  is visited.



# Postorder Traversal



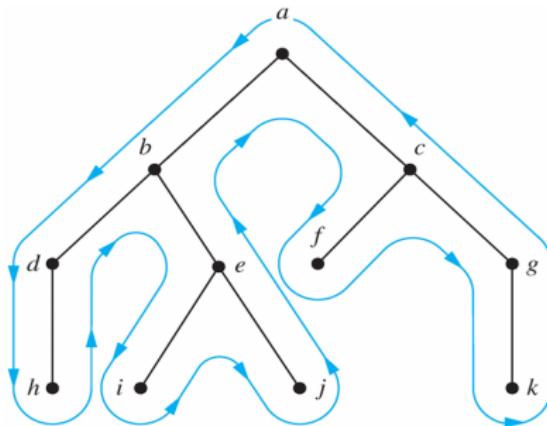
*T*

j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a

# Postorder Traversal: Algorithm

```
procedure postordered ( $T$ : ordered rooted tree)
 $r :=$  root of  $T$ 
for each child  $c$  of  $r$  from left to right
     $T(c) :=$  subtree with  $c$  as root
    postorder( $T(c)$ )
list  $r$ 
```

# Preorder, Inorder, Postorder Traversal



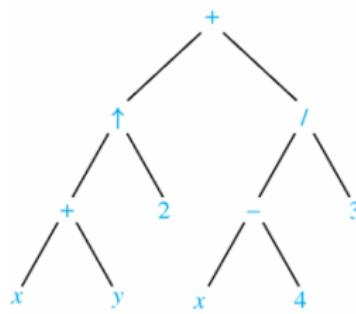
- in preorder: listing each vertex the **first time** this curve passes it.  
a, b, d, h, e, i, j , c, f , g, k
- in inorder: listing a leaf the **first time** the curve passes it and listing each internal vertex the **second time** the curve passes it.  
h, d, b, i, e, j, a, f , c, k, g
- in postorder: listing a vertex the **last time** it is passed on the way back up to its parent.  
h, d, i, j , e, b, f , k, g, c, a

# Expression Trees

Complex expressions can be represented using ordered rooted trees.

- the internal vertices represent operations
- the leaves represent the variables or numbers

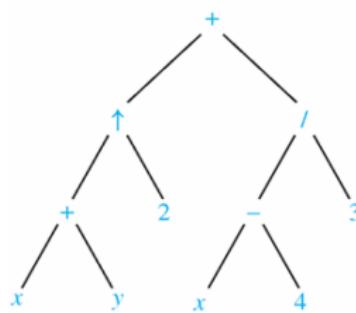
**Example:** Consider the expression  $((x + y) \uparrow 2) + ((x - 4)/3)$



# Infix Notation

An **inorder traversal** of the tree representing an expression produces the original expression when **parentheses are included** except for unary operation. The fully parenthesized expression obtained in this way is said to be in **infix form**.

**Example:**



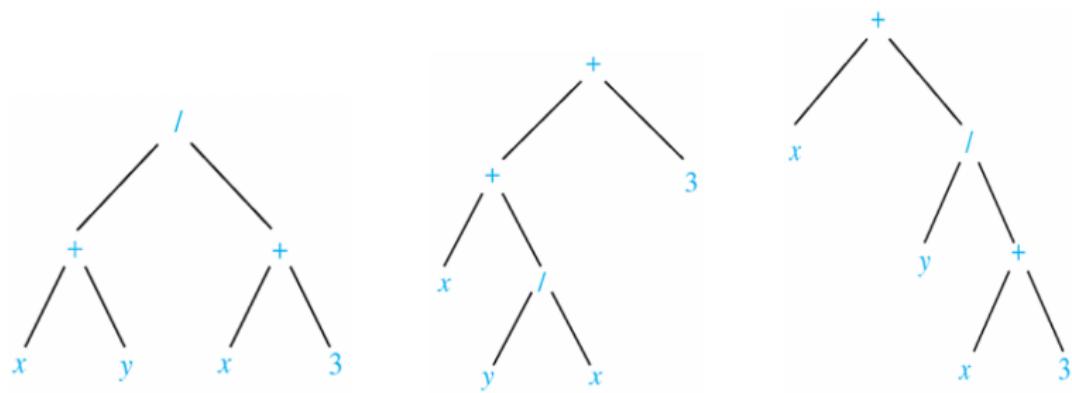
Infix form:  $((x + y) \uparrow 2) + ((x - 4)/3)$

# Infix Notation

An **inorder traversal** of the tree representing an expression produces the original expression when **parentheses are included** except for unary operation. The fully parenthesized expression obtained in this way is said to be in **infix form**.

Why parentheses are needed?

$$(x + y)/(x + 3), (x + (y/x)) + 3, x + (y/(x + 3))$$



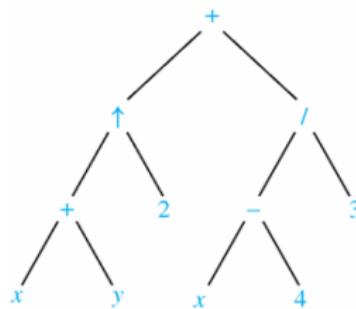
Infix form:  $x + y/x + 3$

# Prefix Notation

The **preorder traversal** of expression trees leads to the **prefix form** of the expression (Polish notation).

An expression in prefix notation, is unambiguous, so **no parentheses** are needed in such an expression.

**Example:** What is the prefix form for  $((x + y) \uparrow 2) + ((x - 4)/3)$ ?



Prefix form:  $+ \uparrow + x y 2 / - x 4 3$ .

# Prefix Notation

Prefix expressions are evaluated by working **from right to left**. When we encounter an operator, we perform the operation with the **two operands to the right**.

**Example:**  $+ - * 2 3 5 / \uparrow 2 3 4$

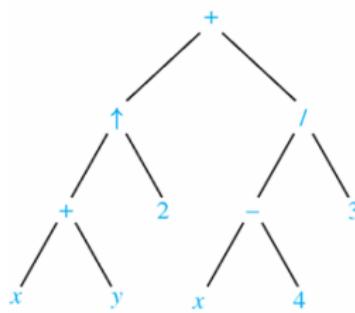
$$\begin{array}{ccccccccc} + & - & * & 2 & 3 & 5 & / & \uparrow & 2 \\ & & & & & & & 2 & 3 \\ & & & & & & & \uparrow & 4 \\ & & & & & & & 2 \uparrow 3 = 8 & \\ + & - & * & 2 & 3 & 5 & / & 8 & 4 \\ & & & & & & & \uparrow & \\ & & & & & & & 8 / 4 = 2 & \\ + & - & * & 2 & 3 & 5 & 2 & & \\ & & & & & & & \uparrow & \\ & & & & & & & 2 * 3 = 6 & \\ + & - & 6 & 5 & 2 & & & \uparrow & \\ & & & & & & & 6 - 5 = 1 & \\ + & 1 & 2 & & & & & \uparrow & \\ & & & & & & & 1 + 2 = 3 & \end{array}$$

# Postfix Notation

The **postorder traversal** of expression trees leads to the **postfix form** of the expression (reverse Polish notation).

Expressions in reverse Polish notation are unambiguous, so parentheses are **not** needed.

**Example:** What is the postfix form of the expression  
 $((x + y) \uparrow 2) + ((x - 4)/3)$ ?



**Postfix form:**  $x\ y\ +\ 2\ \uparrow\ x\ 4\ -\ 3\ /\ +.$

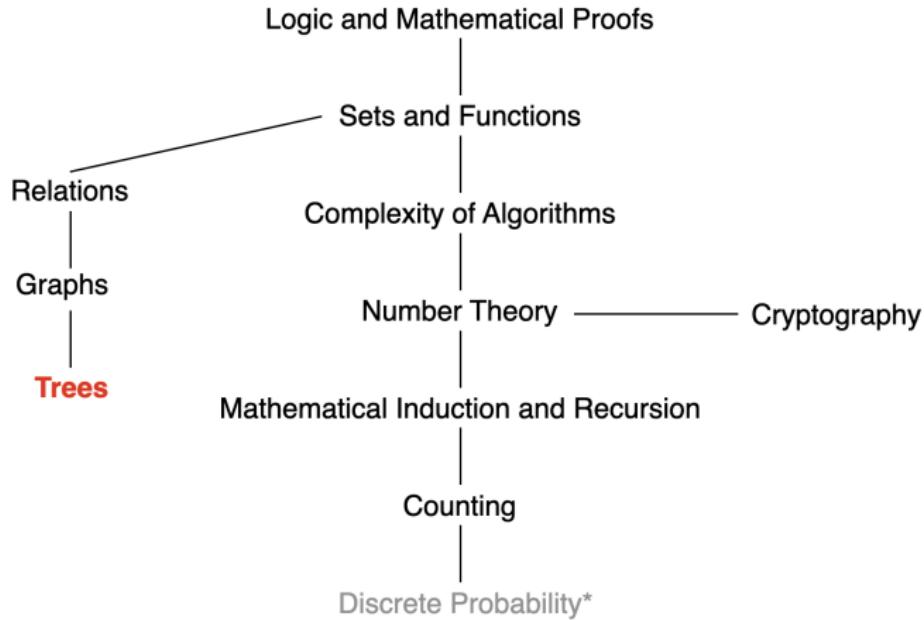
# Postfix Notation

Postfix expressions are evaluated by working **from left to right**. When we encounter an operator, we perform the operation with the **two operands to the left**.

**Example:** 7 2 3 \* - 4 ↑ 9 3 / +

$$\begin{array}{ccccccccc} 7 & 2 & 3 & * & - & 4 & \uparrow & 9 & 3 & / & + \\ \hline & 2 * 3 = 6 & & & & & & & & & \\ 7 & 6 & - & 4 & \uparrow & 9 & 3 & / & + & \\ \hline & 7 - 6 = 1 & & & & & & & & \\ 1 & 4 & \uparrow & 9 & 3 & / & + & & \\ \hline & 1^4 = 1 & & & & & & & & \\ 1 & 9 & 3 & / & + & & & & \\ \hline & 9 / 3 = 3 & & & & & & & & \\ 1 & 3 & + & & & & & & \\ \hline & 1 + 3 = 4 & & & & & & & & \end{array}$$

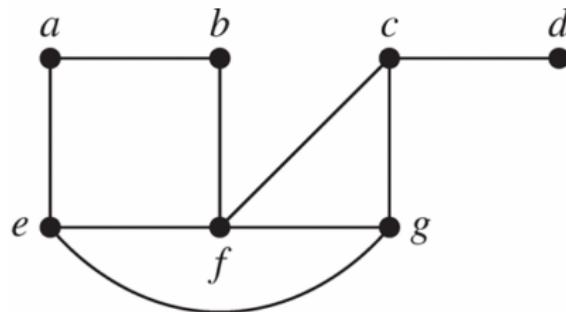
# This Lecture



Tree, Tree Traversal, Spanning Trees ...

# Spanning Trees

**Definition:** Let  $G$  be a simple graph. A **spanning tree** of  $G$  is a **subgraph** of  $G$  that is a tree containing **every** vertex of  $G$ .



Remove edges to **avoid circuits**.

# Spanning Trees

**Theorem** A simple graph is connected if and only if it has a spanning tree.

**Proof:**

- **only if:** The spanning tree can be obtained by removing edges from simple circuits.
- **if:** The spanning tree  $T$  contains every vertex of  $G$ . Furthermore, there is a path in  $T$  between any two of its vertices. Because  $T$  is a subgraph of  $G$ , there is a path in  $G$  between any two of its vertices. Hence,  $G$  is connected.

# Depth-First Search

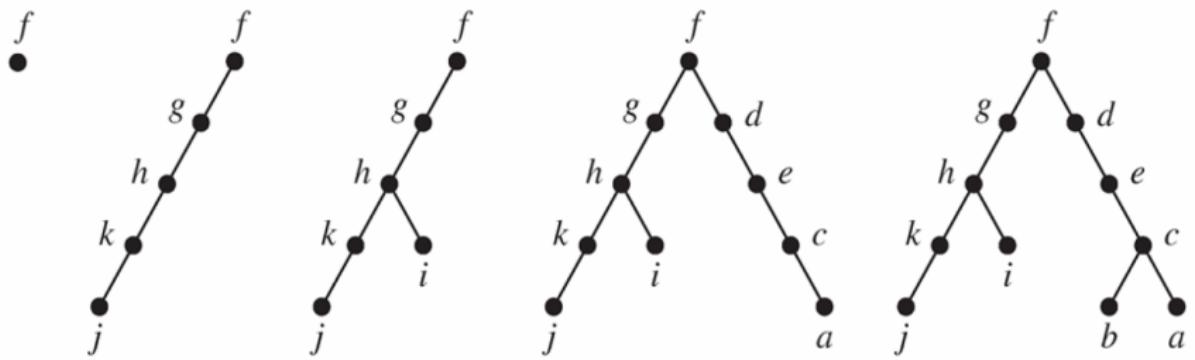
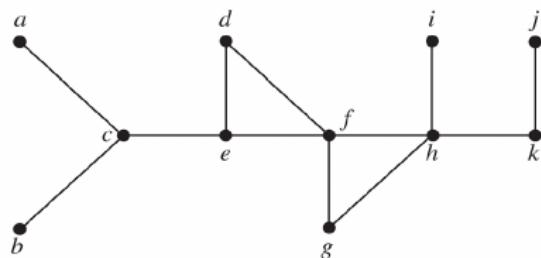
We can find spanning trees by removing edges from simple circuits.

But, this is **inefficient**, since simple circuits should be identified first.

Instead, we build up spanning trees by successively adding edges.

- First, arbitrarily choose a vertex of the graph as the root.
- Form a path by successively **adding vertices** and edges. Continue adding to this path as long as possible.
- If the path goes through **all vertices** of the graph, the tree is a spanning tree.
- Otherwise, move back to some vertex to repeat this procedure (backtracking).

# Depth-First Search: Example



# Depth-First Search: Algorithm

When we add an edge connecting a vertex  $v$  to a vertex  $w$ , we finish exploring from  $w$  before we return to  $v$  to complete exploring from  $v$ .

```
procedure DFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )  
 $T :=$  tree consisting only of the vertex  $v_1$   
 $visit(v_1)$ 
```

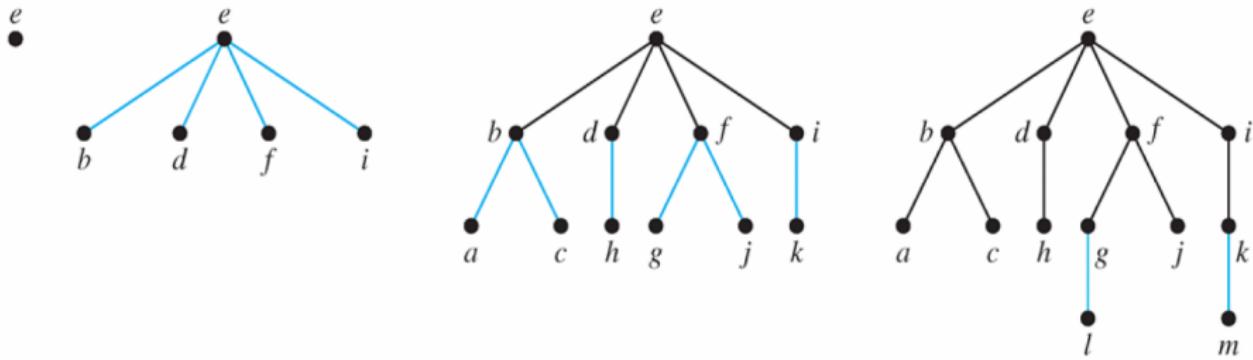
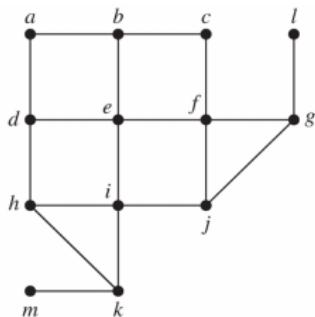
```
procedure visit( $v$ : vertex of  $G$ )  
for each vertex  $w$  adjacent to  $v$  and not yet in  $T$   
    add vertex  $w$  and edge  $\{v, w\}$  to  $T$   
    visit( $w$ )
```

# Breadth-First Search

This is the second algorithm that we build up spanning trees by successively adding edges.

- First arbitrarily choose a vertex of the graph as the root.
- Form a path by adding **all edges** incident to this vertex and the other endpoint of each of these edges
- For each vertex added at the previous level, add edge incident to this vertex, as long as it does not produce a simple circuit.
- Continue in this manner until all vertices have been added.

# Breadth-First Search: Example



# Breadth-First Search

```
procedure BFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )
     $T :=$  tree consisting only of the vertex  $v_1$ 
     $L :=$  empty list  $visit(v_1)$ 
    put  $v_1$  in the list  $L$  of unprocessed vertices
    while  $L$  is not empty
        remove the first vertex,  $v$ , from  $L$ 
        for each neighbor  $w$  of  $v$ 
            if  $w$  is not in  $L$  and not in  $T$  then
                add  $w$  to the end of the list  $L$ 
                add  $w$  and edge  $\{v,w\}$  to  $T$ 
```

# Backtracking Applications

There are problems that can be solved only by performing an **exhaustive search** of all possible solutions.

One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution.

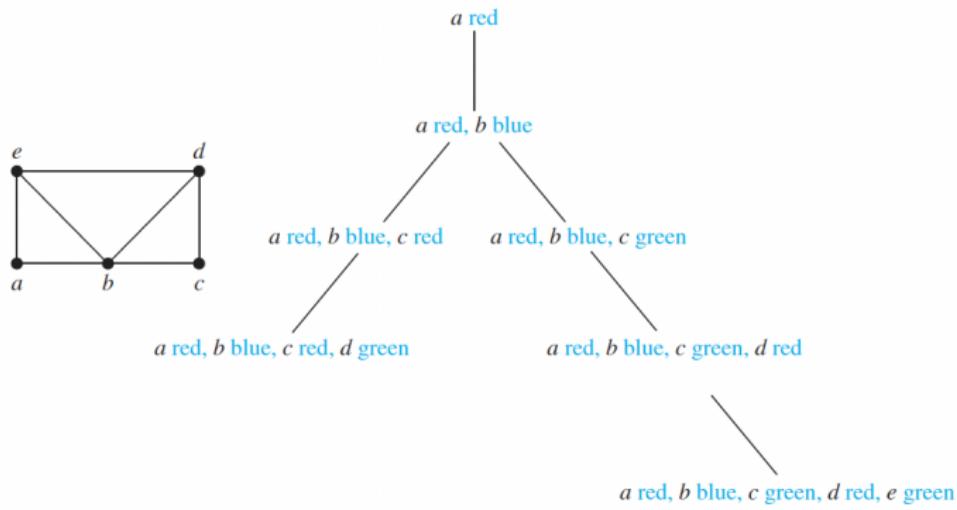
To find a solution via **backtracking**

- first make a sequence of decisions in an attempt to reach a solution as long as this is possible.
- Once it is known that no solution can result from any further sequence of decisions, **backtrack to the parent** of the current vertex and work toward a solution with another series of decisions

The procedure continues until **a solution is found**, or it is established that **no solution** exists.

# Backtracking Applications: Graph Colorings

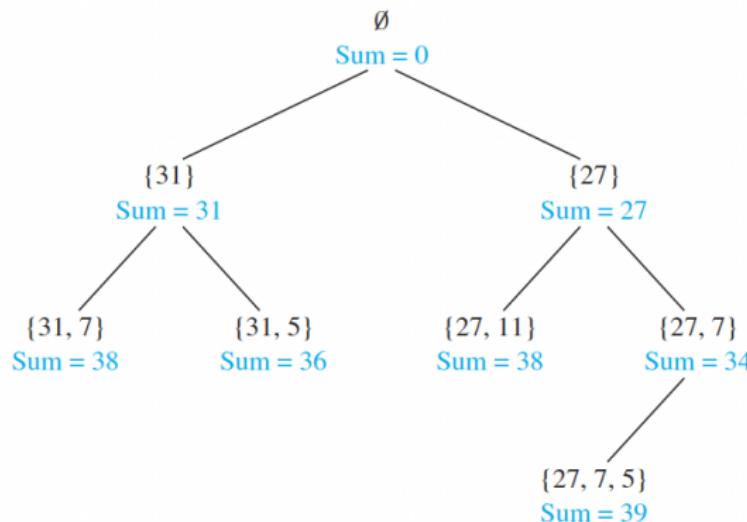
How can backtracking be used to decide whether a graph can be colored using  $n$  colors?



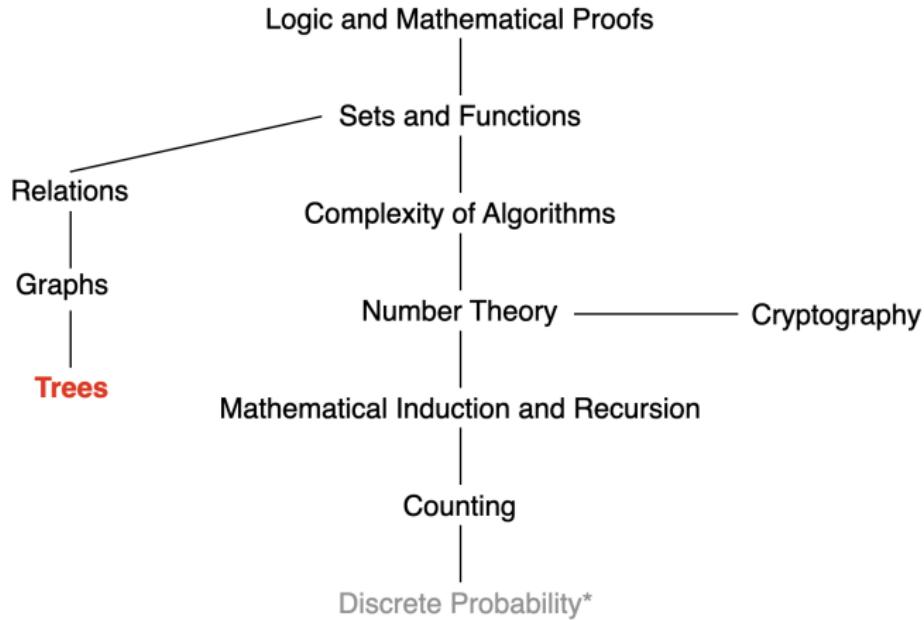
# Backtracking Applications: Sums of Subsets

Consider this problem. Given a set of positive integers  $x_1, x_2, \dots, x_n$ , find a subset of this set of integers that has  $M$  as its sum. How can backtracking be used to solve this problem?

Finding a subset of  $\{31, 27, 15, 11, 7, 5\}$  with the sum equal to 39.



# This Lecture



Tree, Tree Traversal, Spanning Trees, Minimum Spanning Trees, ...



Southern University  
of Science and  
Technology

# Minimum Spanning Trees

**Definition:** A **minimum spanning tree** in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Two greedy algorithms: Prim's Algorithm, Kruscal's Algorithm.

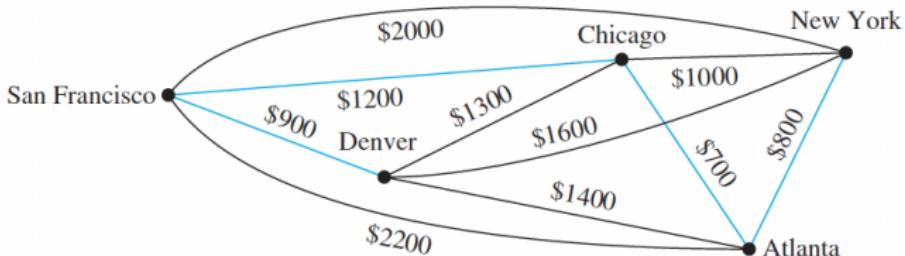
Both algorithms do produce optimal solutions.

# Prim's Algorithm

## ALGORITHM 1 Prim's Algorithm.

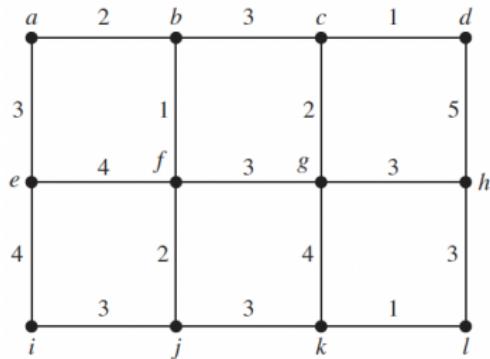
```
procedure Prim(G: weighted connected undirected graph with  $n$  vertices)  
    T := a minimum-weight edge  
    for i := 1 to  $n - 2$   
        e := an edge of minimum weight incident to a vertex in T and not forming a  
            simple circuit in T if added to T  
        T := T with e added  
return T {T is a minimum spanning tree of G}
```

# Prim's Algorithm: Example



| Choice | Edge                     | Cost    |
|--------|--------------------------|---------|
| 1      | {Chicago, Atlanta}       | \$ 700  |
| 2      | {Atlanta, New York}      | \$ 800  |
| 3      | {Chicago, San Francisco} | \$ 1200 |
| 4      | {San Francisco, Denver}  | \$ 900  |
|        | Total:                   | \$ 3600 |

# Prim's Algorithm: Example



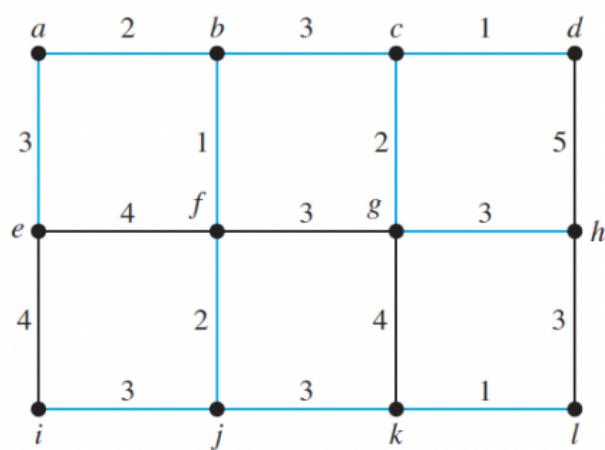
| Choice | Edge     | Weight |
|--------|----------|--------|
| 1      | {b, f}   | 1      |
| 2      | {a, b}   | 2      |
| 3      | {f, j }  | 2      |
| 4      | {a, e}   | 3      |
| 5      | { i, j } | 3      |
| 6      | {f, g}   | 3      |
| 7      | {c, g}   | 2      |
| 8      | {c, d}   | 1      |
| 9      | {g, h}   | 3      |
| 10     | {h, l}   | 3      |
| 11     | {k, l}   | 1      |
| Total: |          | 24     |

# Kruskal's Algorithm

## ALGORITHM 2 Kruskal's Algorithm.

```
procedure Kruskal( $G$ : weighted connected undirected graph with  $n$  vertices)
 $T :=$  empty graph
for  $i := 1$  to  $n - 1$ 
     $e :=$  any edge in  $G$  with smallest weight that does not form a simple circuit
        when added to  $T$ 
     $T := T$  with  $e$  added
return  $T$   $\{T$  is a minimum spanning tree of  $G\}$ 
```

# Kruskal's Algorithm: Example



| Choice | Edge   | Weight    |
|--------|--------|-----------|
| 1      | {c, d} | 1         |
| 2      | {k, l} | 1         |
| 3      | {b, f} | 1         |
| 4      | {c, g} | 2         |
| 5      | {a, b} | 2         |
| 6      | {f, j} | 2         |
| 7      | {b, c} | 3         |
| 8      | {j, k} | 3         |
| 9      | {g, h} | 3         |
| 10     | {i, j} | 3         |
| 11     | {a, e} | 3         |
| Total: |        | <u>24</u> |