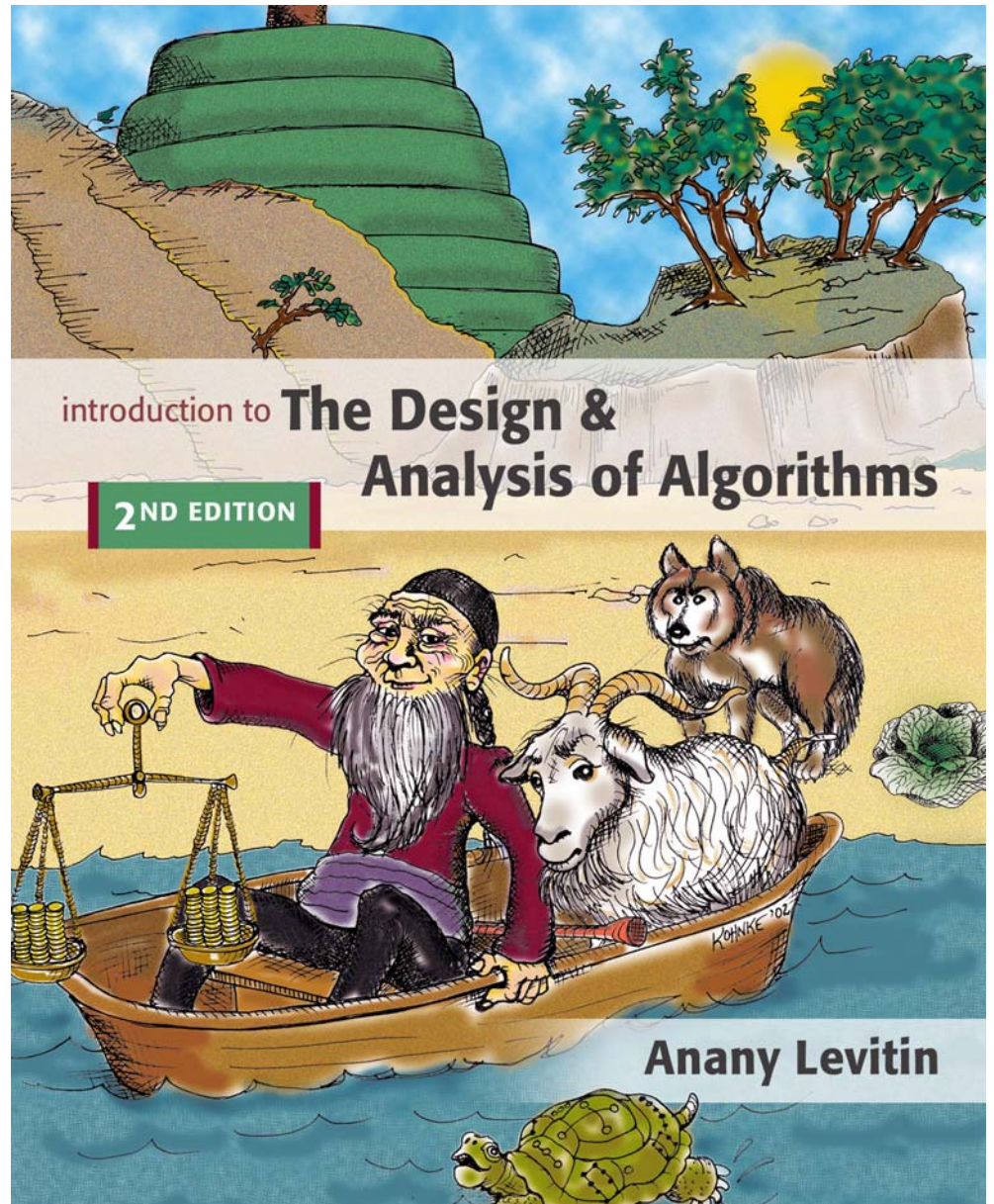


# Chapter 5

## Decrease-and-Conquer



# Decrease-and-Conquer



- 1. Reduce problem instance to smaller instance of the same problem**
  - 2. Solve smaller instance**
  - 3. Extend solution of smaller instance to obtain solution to original instance**
- ∩ **Can be implemented either top-down or bottom-up**
- ∩ **Also referred to as *inductive* or *incremental* approach**

# 3 Types of Decrease and Conquer



## $\Omega$ Decrease by a constant (usually by 1):

- insertion sort
- graph traversal algorithms (DFS and BFS)
- topological sorting
- algorithms for generating permutations, subsets

## $\Omega$ Decrease by a constant factor (usually by half)

- binary search and bisection method
- exponentiation by squaring
- multiplication à la russe

## $\Omega$ Variable-size decrease

- Euclid's algorithm
- selection by partition
- Nim-like games

# What's the difference?



Consider the problem of exponentiation: Compute  $a^n$

- ⌚ **Brute Force:**
- ⌚ **Divide and conquer:**
- ⌚ **Decrease by one:**
- ⌚ **Decrease by constant factor:**

# Insertion Sort



To sort array  $A[0..n-1]$ , sort  $A[0..n-2]$  recursively and then insert  $A[n-1]$  in its proper place among the sorted  $A[0..n-2]$

Ω Usually implemented bottom up (nonrecursively)

**Example: Sort 6, 4, 1, 8, 5**

|   |   |          |          |          |          |
|---|---|----------|----------|----------|----------|
| 6 |   | <u>4</u> | 1        | 8        | 5        |
| 4 | 6 |          | <u>1</u> | 8        | 5        |
| 1 | 4 | 6        |          | <u>8</u> | 5        |
| 1 | 4 | 6        | 8        |          | <u>5</u> |
| 1 | 4 | 5        | 6        | 8        |          |

# Pseudocode of Insertion Sort

**ALGORITHM** *InsertionSort*( $A[0..n - 1]$ )

//Sorts a given array by insertion sort

//Input: An array  $A[0..n - 1]$  of  $n$  orderable elements

//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order

**for**  $i \leftarrow 1$  **to**  $n - 1$  **do**

$v \leftarrow A[i]$

$j \leftarrow i - 1$

**while**  $j \geq 0$  **and**  $A[j] > v$  **do**

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

# Analysis of Insertion Sort



## ⌚ Time efficiency

$$C_{\text{worst}}(n) = n(n-1)/2 \in \Theta(n^2)$$

$$C_{\text{avg}}(n) \approx n^2/4 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = n - 1 \in \Theta(n) \quad (\text{also fast on almost sorted arrays})$$

## ⌚ Space efficiency: in-place

## ⌚ Stability: yes

## ⌚ Best elementary sorting algorithm overall

## ⌚ Binary insertion sort



# Graph Traversal



**Many problems require processing all graph vertices (and edges) in systematic fashion**

## **Graph traversal algorithms:**

- **Depth-first search (DFS)**
- **Breadth-first search (BFS)**



# Depth-First Search (DFS)



- ∞ **Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.**
  
- ∞ **Uses a stack**
  - **a vertex is pushed onto the stack when it's reached for the first time**
  - **a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex**
  
- ∞ **“Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)**

# Pseudocode of DFS



## ALGORITHM *DFS(G)*

//Implements a depth-first search traversal of a given graph

//Input: Graph  $G = \langle V, E \rangle$

//Output: Graph  $G$  with its vertices marked with consecutive integers

//in the order they've been first encountered by the DFS traversal

mark each vertex in  $V$  with 0 as a mark of being “unvisited”

*count*  $\leftarrow$  0

**for** each vertex  $v$  in  $V$  **do**

**if**  $v$  is marked with 0

*dfs(v)*

*dfs(v)*

//visits recursively all the unvisited vertices connected to vertex  $v$  by a path

//and numbers them in the order they are encountered

//via global variable *count*

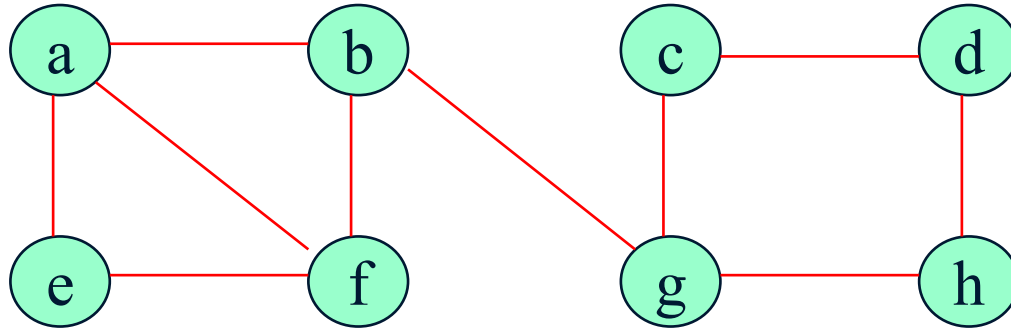
*count*  $\leftarrow$  *count* + 1; mark  $v$  with *count*

**for** each vertex  $w$  in  $V$  adjacent to  $v$  **do**

**if**  $w$  is marked with 0

*dfs(w)*

# Example: DFS traversal of undirected graph



**DFS traversal stack:**

**DFS tree:**

# Notes on DFS



∞ **DFS can be implemented with graphs represented as:**

- **adjacency matrices:  $\Theta(V^2)$**
- **adjacency lists:  $\Theta(|V|+|E|)$**

∞ **Yields two distinct ordering of vertices:**

- **order in which vertices are first encountered (pushed onto stack)**
- **order in which vertices become dead-ends (popped off stack)**

∞ **Applications:**

- **checking connectivity, finding connected components**
- **checking acyclicity**
- **finding articulation points and biconnected components**
- **searching state-space of problems for solution (AI)**

# Breadth-first search (BFS)



- ⌚ Visits graph vertices by moving across to all the neighbors of last visited vertex
- ⌚ Instead of a stack, BFS uses a queue
- ⌚ Similar to level-by-level tree traversal
- ⌚ “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

# Pseudocode of BFS

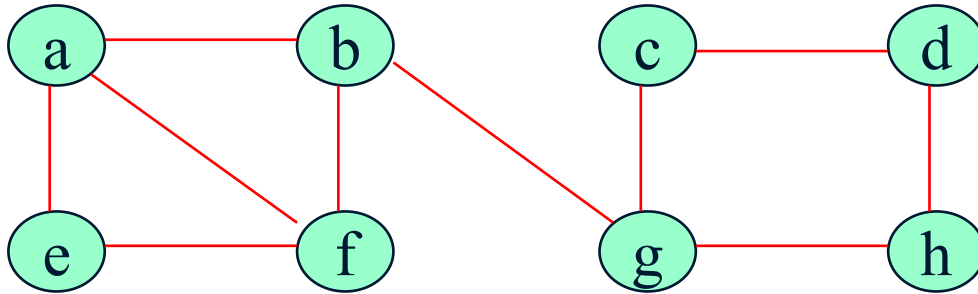


## ALGORITHM *BFS*(*G*)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph  $G = \langle V, E \rangle$ 
//Output: Graph  $G$  with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in  $V$  with 0 as a mark of being “unvisited”
count  $\leftarrow$  0
for each vertex  $v$  in  $V$  do
    if  $v$  is marked with 0
        bfs( $v$ )

bfs( $v$ )
//visits all the unvisited vertices connected to vertex  $v$  by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count  $\leftarrow$  count + 1; mark  $v$  with count and initialize a queue with  $v$ 
while the queue is not empty do
    for each vertex  $w$  in  $V$  adjacent to the front vertex do
        if  $w$  is marked with 0
            count  $\leftarrow$  count + 1; mark  $w$  with count
            add  $w$  to the queue
    remove the front vertex from the queue
```

# Example of BFS traversal of undirected graph



**BFS traversal queue:**

**BFS tree:**



# Notes on BFS

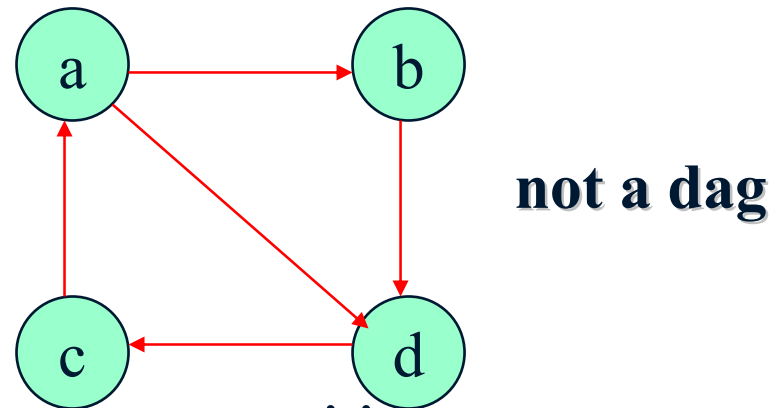
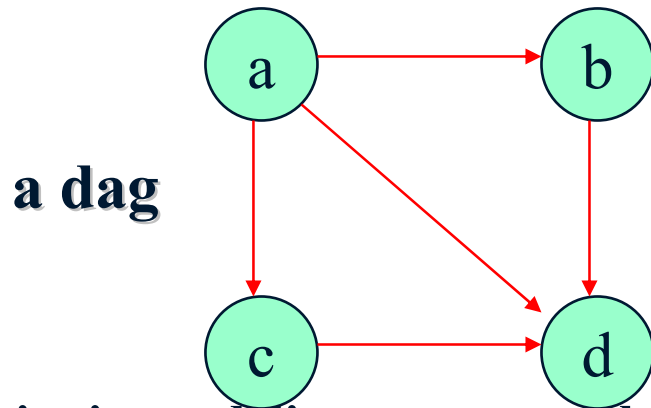


- ⌚ **BFS has same efficiency as DFS and can be implemented with graphs represented as:**
  - **adjacency matrices:  $\Theta(V^2)$**
  - **adjacency lists:  $\Theta(|V|+|E|)$**
- ⌚ **Yields single ordering of vertices (order added/deleted from queue is the same)**
- ⌚ **Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges**

# Dags and Topological Sorting



A ***dag***: a directed acyclic graph, i.e. a directed graph with no (directed) cycles



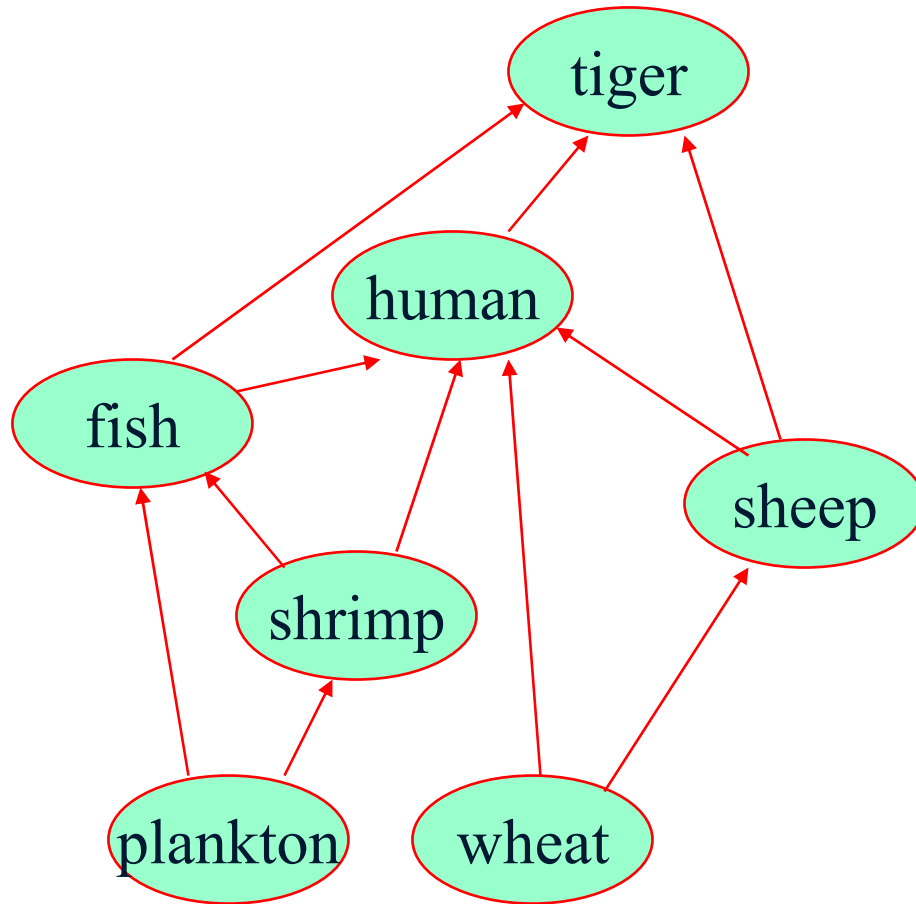
Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (***topological sorting***). Being a dag is also a necessary condition for topological sorting be possible.

# Topological Sorting Example



Order the following items in a food chain



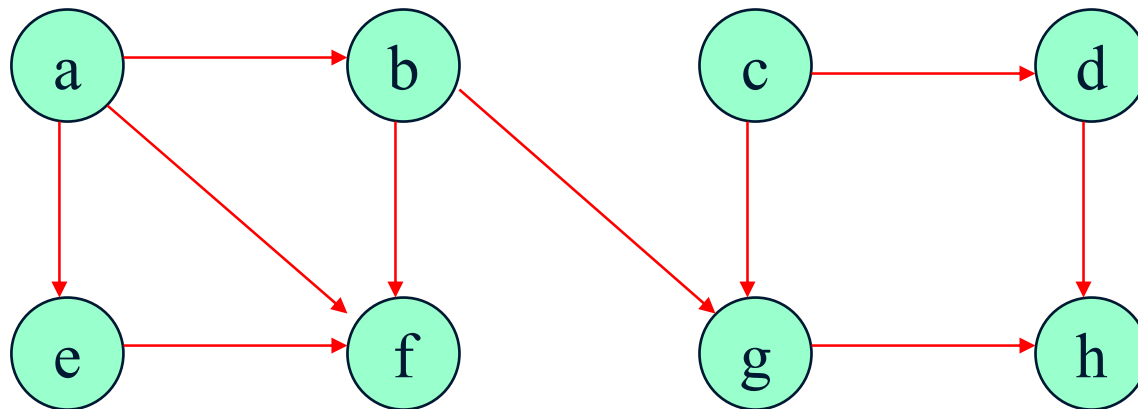
# DFS-based Algorithm



## DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered? → NOT a dag!

**Example:**



**Efficiency:**

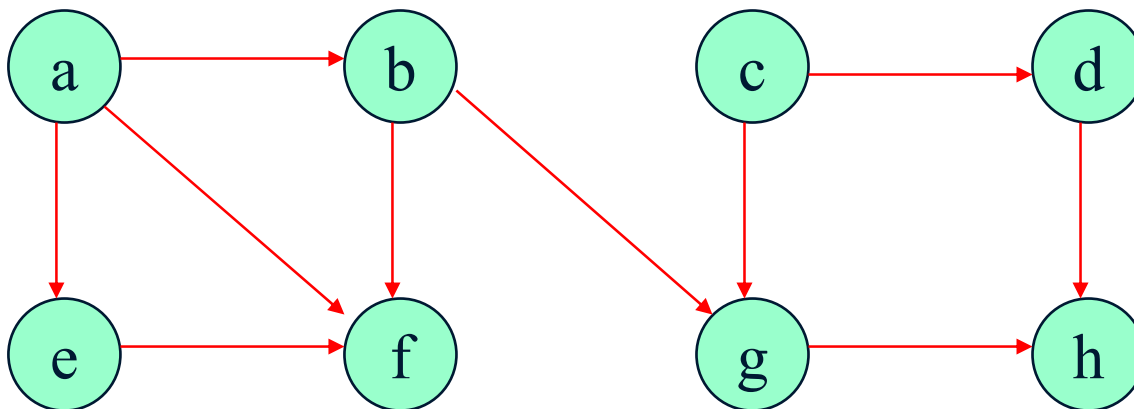
# Source Removal Algorithm



## Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

**Example:**




**Efficiency: same as efficiency of the DFS-based algorithm**

# Decrease-by-Constant-Factor Algorithms



**In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)**

## **Examples:**

- **Binary search and the method of bisection**
  - **Exponentiation by squaring**
  - **Multiplication à la russe (Russian peasant method)**
  - **Fake-coin puzzle**
  - **Josephus problem**
- 

# Exponentiation by Squaring



**The problem: Compute  $a^n$  where  $n$  is a nonnegative integer**

**The problem can be solved by applying recursively the formulas:**

**For even values of  $n$**

$$a^n = (a^{n/2})^2 \text{ if } n > 0 \text{ and } a^0 = 1$$

**For odd values of  $n$**

$$a^n = (a^{(n-1)/2})^2 a$$

**Recurrence:  $M(n) = M(\lfloor n/2 \rfloor) + f(n)$ , where  $f(n) = 1$  or  $2$ ,**

$$M(0) = 0$$

**Master Theorem:  $M(n) \in \Theta(\log n) = \Theta(b)$  where  $b = \lceil \log_2(n+1) \rceil$**



# Russian Peasant Multiplication



**The problem: Compute the product of two positive integers**

**Can be solved by a decrease-by-half algorithm based on the following formulas.**

**For even values of  $n$ :**

$$n * m = \frac{n}{2} * 2m$$

**For odd values of  $n$ :**

$$n * m = \frac{n-1}{2} * 2m + m \text{ if } n > 1 \text{ and } m \text{ if } n = 1$$

# Example of Russian Peasant Multiplication



Compute  $20 * 26$

| <i>n</i> | <i>m</i> |       |
|----------|----------|-------|
| 20       | 26       |       |
| 10       | 52       |       |
| 5        | 104      | 104   |
| 2        | 208      | +     |
| 1        | 416      | 416   |
|          |          | <hr/> |
|          |          | 520   |

**Note: Method reduces to adding  $m$ 's values corresponding to odd  $n$ 's.**

# Fake-Coin Puzzle (simpler version)



**There are  $n$  identically looking coins one of which is fake. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much). Design an efficient algorithm for detecting the fake coin. Assume that the fake coin is known to be lighter than the genuine ones.**

**Decrease by factor 2 algorithm**

**Decrease by factor 3 algorithm**

# Variable-Size-Decrease Algorithms



**In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another**

## **Examples:**

- **Euclid's algorithm for greatest common divisor**
- **Partition-based algorithm for selection problem**
- **Interpolation search**
- **Some algorithms on binary search trees**
- **Nim and Nim-like games**

# Euclid's Algorithm



**Euclid's algorithm is based on repeated application of equality**

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

**Ex.:  $\gcd(80, 44) = \gcd(44, 36) = \gcd(36, 12) = \gcd(12, 0) = 12$**

**One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations.**

**Hence,  $T(n) \in O(\log n)$**

# Selection Problem



Find the  $k$ -th smallest element in a list of  $n$  numbers

⌚  $k = 1$  or  $k = n$

⌚ median:  $k = \lceil n/2 \rceil$

**Example: 4, 1, 10, 9, 7, 12, 8, 2, 15    median = ?**

**The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.**

# Digression: Post Office Location Problem



**Given  $n$  village locations along a straight highway, where should a new post office be located to minimize the average distance from the villages to the post office?**



# Algorithms for the Selection Problem



**The sorting-based algorithm: Sort and return the  $k$ -th element**  
**Efficiency (if sorted by mergesort):  $\Theta(n \log n)$**

**A faster algorithm is based on using the quicksort-like partition of the list.**  
**Let  $s$  be a split position obtained by a partition:**

all are  $\leq A[s]$

all are  $\geq A[s]$

**Assuming that the list is indexed from 1 to  $n$ :**

**If  $s = k$ , the problem is solved;**

**if  $s > k$ , look for the  $k$ -th smallest elem. in the left part;**

**if  $s < k$ , look for the  $(k-s)$ -th smallest elem. in the right part.**

**Note: The algorithm can simply continue until  $s = k$ .**

# Tracing the Median / Selection Algorithm



**Example:** 4 1 10 9 7 12 8 2 15      Here:  $n = 9, k = \lceil 9/2 \rceil = 5$

| array index | 1        | 2 | 3        | 4        | 5        | 6        | 7  | 8  | 9  |                 |
|-------------|----------|---|----------|----------|----------|----------|----|----|----|-----------------|
|             | <u>4</u> | 1 | 10       | 9        | 7        | 12       | 8  | 2  | 15 |                 |
|             | <u>4</u> | 1 | 2        | 9        | 7        | 12       | 8  | 10 | 15 |                 |
|             | 2        | 1 | <u>4</u> | 9        | 7        | 12       | 8  | 10 | 15 | --- $s=3 < k=5$ |
|             |          |   |          | <u>9</u> | 7        | 12       | 8  | 10 | 15 |                 |
|             |          |   |          | <u>9</u> | 7        | 8        | 12 | 10 | 15 |                 |
|             |          |   |          | 8        | 7        | <u>9</u> | 12 | 10 | 15 | --- $s=6 > k=5$ |
|             |          |   |          | <u>8</u> | 7        |          |    |    |    |                 |
|             |          |   |          | 7        | <u>8</u> |          |    |    |    | --- $s=k=5$     |

**Solution: median is 8**

# Efficiency of the Partition-based Algorithm



**Average case (average split in the middle):**

$$C(n) = C(n/2) + (n+1)$$

$$C(n) \in \Theta(n)$$

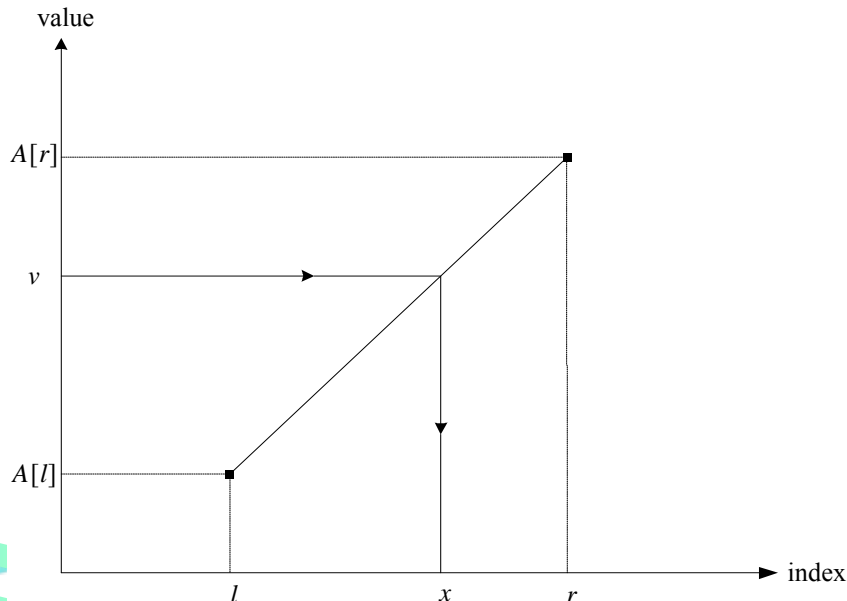
**Worst case (degenerate split):**  $C(n) \in \Theta(n^2)$

**A more sophisticated choice of the pivot leads to a complicated algorithm with  $\Theta(n)$  worst-case efficiency.**

# Interpolation Search



Searches a sorted array similar to binary search but estimates location of the search key in  $A[l..r]$  by using its value  $v$ . Specifically, the values of the array's elements are assumed to grow linearly from  $A[l]$  to  $A[r]$  and the location of  $v$  is estimated as the  $x$ -coordinate of the point on the straight line through  $(l, A[l])$  and  $(r, A[r])$  whose  $y$ -coordinate is  $v$ :



$$x = l + \lfloor (v - A[l])(r - l) / (A[r] - A[l]) \rfloor$$

# Analysis of Interpolation Search



## ∞ Efficiency

average case:  $C(n) < \log_2 \log_2 n + 1$

worst case:  $C(n) = n$

∞ Preferable to binary search only for **VERY** large arrays and/or expensive comparisons

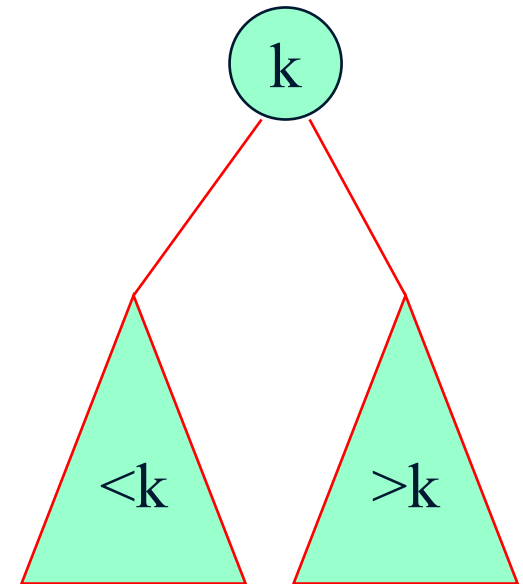
∞ Has a counterpart, the method of false position (regula falsi), for solving equations in one unknown (Sec. 12.4)

# Binary Search Tree Algorithms



Several algorithms on BST requires recursive processing of just one of its subtrees, e.g.,

- ∞ Searching
- ∞ Insertion of a new key
- ∞ Finding the smallest (or the largest) key



# Searching in Binary Search Tree



**Algorithm**  $BTS(x, v)$

//Searches for node with key equal to  $v$  in BST rooted at node  $x$

**if**  $x = \text{NIL}$  **return** -1

**else if**  $v = K(x)$  **return**  $x$

**else if**  $v < K(x)$  **return**  $BTS(\text{left}(x), v)$

**else return**  $BTS(\text{right}(x), v)$

**Efficiency**

**worst case:**  $C(n) = n$

**average case:**  $C(n) \approx 2 \ln n \approx 1.39 \log_2 n$



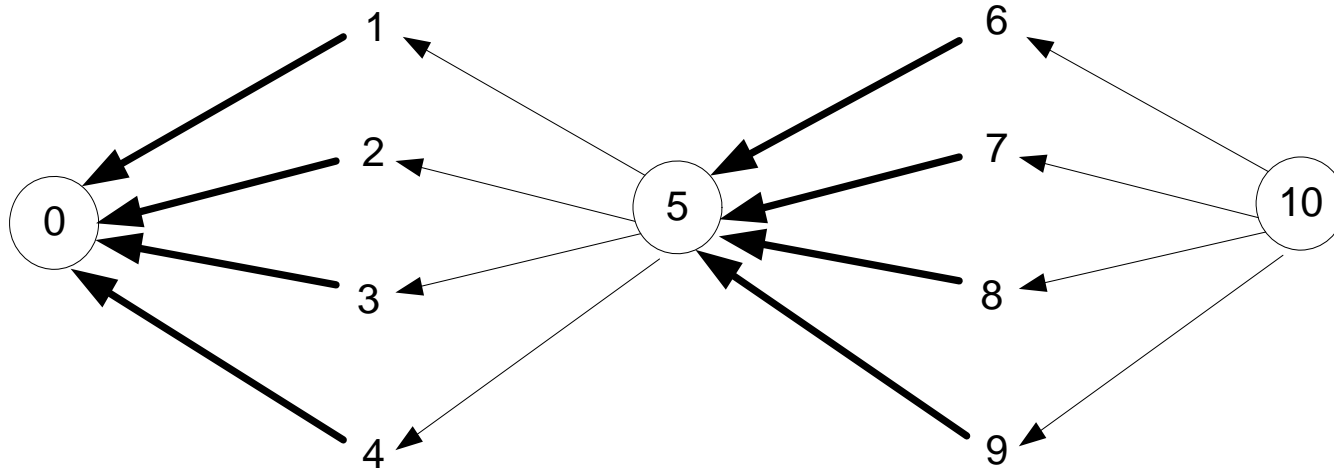
# One-Pile Nim



**There is a pile of  $n$  chips. Two players take turn by removing from the pile at least 1 and at most  $m$  chips. (The number of chips taken can vary from move to move.) The winner is the player that takes the last chip. Who wins the game – the player moving first or second, if both player make the best moves possible?**

**It's a good idea to analyze this and similar games “backwards”, i.e., starting with  $n = 0, 1, 2, \dots$**

# Partial Graph of One-Pile Nim with $m = 4$



**Vertex numbers indicate  $n$ , the number of chips in the pile. The losing position for the player to move are circled. Only winning moves from a winning position are shown (in bold).**

**Generalization: The player moving first wins iff  $n$  is not a multiple of 5 (more generally,  $m+1$ ); the winning move is to take  $n \bmod 5$  ( $n \bmod (m+1)$ ) chips on every move.**