Introduction:
Data Structures and Algorithms

성균관대학교 컴퓨터공학과
데이터베이스 연구실
김응모
Algorithm

Definition of Algorithm

Algorithm is a step-by-step procedure for solving a problem in a finite amount of time. It consists of:
- Instructions
- Input data
- Output data

Classes of Algorithms: Computer Science
- Searching algorithms
- Sorting algorithms
- Tree algorithms
- Graph algorithms
- Hashing algorithms
- Parsing algorithms
- ........
Example: Algorithm

- Problem: Compute GCD
- Algorithm: Euclidean Algorithm
  - Input: Integers \((L \geq S)\)
  - Output: GCD of \(L, S\)

```c
GCD (int L, S)
    int R;
    while (S > 0)
        {
            R = L % S;
            L = S;
            S = R;
        }
    return (L)
```
Problem: Compute $X^N$

Algorithm: Power Algorithm

- Input: Integers $X$, $N$
- Output: $X^N$

```c
POWER (int X, int N)
    if (N == 0) return 1;
    else {
        factor = POWER(X, N/2)
        if  N%2 == 0  return factor*factor
        else  return factor*factor*X
    } 
```
Example: Algorithm

- Problem: Sorting integers
- Algorithm: Selection Sort
  - Input: n unsorted integers
  - Output: n sorted integers

**Selection Sort** (int list[n])

```c
for (i = 0; i < n; i++)
{
    1) Examine list[i] to list[n-1]
    2) Find the smallest integer
    3) Let it store list[min];
    4) Swap list[i] and list[min]
}
```
Performance Analysis

◆ **Performance Analysis**

- **Space Complexity**
  - the amount of memory space used by the algorithm

- **Time Complexity**
  - the amount of computing time used by the algorithm

◆ Typically, the more (less) space, the less (more) time.
   
   Thus, sometimes we need to trade off space vs. time.
Space Complexity

- Find a total sum of n numbers. Space = ?

```c
SUM (float list[], int n)
sum = 0;
int i;
for (i = 0; i < n; i++)
    sum = sum + list[i];
return sum;
```

- Addition of two n x n matrices. Space = ?

- Representing an n x n sparse matrix. Space = ?
Time Complexity

- Time Complexity Criteria?
  - Theoretical Speed
    - number of operations by performed by the algorithm.
  - Practical Speed
    - the execution time performed by the algorithm.

```c
sum = 0;
for (i = 0;  i < 1000000; i++)
  sum = sum + i ;
```

- What is time complexity?
  - Theoretical Speed : $10^6$ (additions)
  - Practical Speed : 10 msec. (Assume: Pentium III, 256M memory)

- Which criteria is more reasonable?
  - “Theoretical” speed gives better criteria. Why?
### Time Complexity

- **Linear**
  ```
  for (i=1; i<=n; i++)
      { application code }
  Time = ?
  ```
  ```
  for (i=1; i<=n; i+=2)
      { application code }
  Time = ?
  ```

- **Logarithmic**
  ```
  for (i=1; i<=n; i*= 2)
      { application code }
  Time = ?
  ```
  ```
  for (i=n; i>=1; i/=2)
      { application code }
  Time = ?
  ```

- **Quadratic**
  ```
  for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
          { application code }
  Time = ?
  ```
  ```
  for (i=1; i<=n; i++)
      for (j=1; j<=i; j++)
          { application code }
  Time = ?
  ```

- **Dependent quadratic**
  ```
  for (i=1; i<=n; i++)
      for (j=1; j<=i; j++)
          { application code }
  Time = ?
  ```

- **Linear logarithmic**
  ```
  for (i=1; i<=n; i++)
      for (j=1; j<=n; j*=2)
          { application code }
  Time = ?
  ```
  ```
  for (i=1; i<=n; i++)
      for (j=1; j<=n; j*=2)
          { application code }
  Time = ?
  ```
Time Performances : Big Oh (O)

- Which one is faster?
  Example:

  ![Diagram showing algorithms A and B with functions f(n) = 10n and g(n) = 1/2n^2]

- Given f(n) and g(n), we say that f(n) = \( O(g(n)) \) if there are positive constants c and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

- Note: c is implementation factor depending on H/W and S/W environmental variants. If \( f(n) = a_k n^k + \ldots + a_1 n + a_0 \), then \( f(n) = O(n^k) \).
Class of Time Complexities

◆ Polynomial Time
  ▪ O(1) : Constant
  ▪ O(log₂n)
  ▪ O(n)
  ▪ O(n·log₂n)
  ▪ O(n²)
  ▪ O(n³)
  ...... 
  ▪ O(n^k)

◆ Exponential Time
  ▪ O(2^n)
  ▪ O(n!)
  ▪ O(n^n)
Class of Time Complexities

◆ Which one is bigger?
  - O(n^k) vs O(2^n)
  - O(n^k) : Easy, Reasonable, Mostly solved within by O(n^3)
  - O(2^n) : Hard, Cannot be solved in practice.

◆ Ordering of complexities
  - O(1) < O(log_2 n) < O(n) < O(n log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)

◆ Which are meaning of these comparisons?
  - O(n) vs O(1)
  - O(n) vs O(log_2 n)
  - O(n^2) vs O(n log_2 n)
  - O(n^3) vs O(n^2)
# Growth of Function Values

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^2$</td>
<td>1.7 mins</td>
</tr>
<tr>
<td>$10^3$</td>
<td>17 mins</td>
</tr>
<tr>
<td>$10^4$</td>
<td>2.8 hrs</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.1 days</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>$10^7$</td>
<td>3.8 months</td>
</tr>
<tr>
<td>$10^8$</td>
<td>3.1 years</td>
</tr>
<tr>
<td>$10^9$</td>
<td>3.1 decades</td>
</tr>
</tbody>
</table>

## Powers of 2

- $2^{10} = 10^3$
- $2^{20} = 10^6$
- $2^{30} = 10^9$

## Logarithmic

- $\log_2 10^3 = 10$
- $\log_2 10^6 = 20$
- $\log_2 10^9 = 30$

<table>
<thead>
<tr>
<th>Time for $f(n)$ instructions on a $10^9$ instr/sec computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>1,000</td>
</tr>
<tr>
<td>10,000</td>
</tr>
<tr>
<td>100,000</td>
</tr>
<tr>
<td>1,000,000</td>
</tr>
</tbody>
</table>
Example: Sorting

(35 38 70 75 12 25 18 54 65 90 86)

\[ \text{sorting} \]

(12 18 25 35 38 54 65 70 75 86 90) : sorted

- Classic Problem in Computer Science: Still many researches!
- Sorting is essential for solving many problems efficiently.
- 25% ~ 50 of total time for solving problem is spent for sorting.
- Performance Criteria: Number of Comparisons
- Selection, Bubble, Insertion, Heap, Shell, Quick, Merge
- $O(n^2)$ or $O(n\log_2 n)$
Sorting

How many comparison operations? (Input size n)

- Selection Sort
- Bubble Sort
- Insertion Sort
- Quick Sort
- Merge Sort

<table>
<thead>
<tr>
<th>list:</th>
<th>26</th>
<th>5</th>
<th>37</th>
<th>1</th>
<th>61</th>
<th>11</th>
<th>59</th>
<th>15</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Comparison: Sorting Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Average</th>
<th>Worst</th>
<th>Extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bubble</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n\log_2 n)$</td>
<td>$O(n^2)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n\log_2 n)$</td>
<td>$O(n\log_2 n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- Insertion sort is the best for small $n$.
- Quick sort is the best in average case.
- Merge sort is the best in worst case, but we need extra space.
- We usually combine Insertion, Quick, and Merge.
### Sorting: Performance

- **Algorithms by Sedgewick**
  - ✓ PC: $10^8$ comparisons/sec
  - ✓ Super: $10^{12}$ comparisons/sec

<table>
<thead>
<tr>
<th>Insertion Sort (O(n^2))</th>
<th>n = 10^3</th>
<th>n = 10^6</th>
<th>n = 10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>instant</td>
<td>instant</td>
<td>2.8hrs</td>
</tr>
<tr>
<td>Super</td>
<td>instant</td>
<td>1sec</td>
<td>instant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Merge Sort (O(nlog_2n))</th>
<th>n = 10^3</th>
<th>n = 10^6</th>
<th>n = 10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>instant</td>
<td>1sec</td>
<td>18min</td>
</tr>
<tr>
<td>Super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quick Sort (O(nlog_2n))</th>
<th>n = 10^3</th>
<th>n = 10^6</th>
<th>n = 10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>instant</td>
<td>0.3sec</td>
<td>6min</td>
</tr>
<tr>
<td>Super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

- Good algorithms are better than supercomputers.
- Good algorithms are better than good ones.
Practical Complexities

- Sequential Search: \( O(n) \)
- Binary Search: \( O(\log_2 n) \)
- External (B-Tree) Search: \( O(\log_f n), f \approx 133 \)
- Selection, Bubble, Insertion Sort: \( O(n^2) \)
- Quick, Heap, Merge Sort: \( O(n \log_2 n) \)
- Euler Cycle: \( O(n^2) \)
- Minimal Spanning Tree: \( O(n \log_2 n) \)
- Shortest Paths: \( O(n^2) \)
- Matrix Addition: \( O(n^2) \)
- Matrix Multiplication: \( O(n^3) \) or \( O(n^{2.81}) \)
- Satisfiability Problem: \( O(2^n) \)
- Hamiltonian Cycle: \( O(n!) \)
- Graph Coloring: \( O(n^n) \)

.............
How do we store the following data in memory efficiently?

- Matrix Operations
- Mazing Problem
- Bank Customers Service
- UNIX File Directory
- Baseball Tournament
- Airline Flights Connection
- Given n integers, find an arbitrary number?
- Given n integers, find a maximum number?
- Courses Road Map
- ........
Data Structures

Data Structure
- How do we store data in a (mostly) memory?
- We need to specify data structure to organize them.
- Choice of different data structures gives us different algorithms.
- Good data structures are essential for efficient algorithms.

Memory

Data Structures

View

Implement

(a) Matrix
(b) Linear list
(c) Tree
(d) Graph
Array/Linked List

**Array**
- A linear list with (index, value)
- Consecutive memory locations
- Static Allocation : Compile Time
- Reads/writes : O(1)
- Insert/deletes : O(n)

**Linked List**
- A linear list with pointers(links)
- Non-Consecutive memory locations
- Dynamic Allocation : Run Time
- Reads/writes : O(1)
- Insert/deletes : O(1)
Two Approaches: Arrays vs Linked Lists

- Lists (1 dimension): Searching, Sorting, . . .
- Matrix Operations
- Binary Trees: Especially, Complete binary tree
- Trees
- Heaps
- Graphs: Roads, Maps, SNS Networks, . . .
Array : Sparse Matrix

Sparse Matrix : Most elements are 0’s; Real values are rare.
Examples : Airline Flights, Web Pages Matrix, . . .

(Conventional) 2-D array

- $A[m, n] \ (m : \#\text{rows}, \ n : \#\text{columns})$
- Memory Usage : $t / (m \times n) \ (t : \#\text{non-zeros})$
- Very inefficient!
Array : Sparse Matrix

<table>
<thead>
<tr>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-15</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>-6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>28</td>
</tr>
</tbody>
</table>

- Compressed 2-D array
  - Stores only non-zero values; By raw-major order;
  - <row position, column position, non-zero value>
  - Memory Usage : $\propto t$ (independent of matrix size)
  - Efficient!
Stack

- A linear List with top and bottom.
- All insertions and deletions occur at top.
- Push(insert) and Pop(delete)
- Top values grow and shrink.
- All items except top are invisible.
- LIFO (Last-In First-Out)
Implementing Stack

◆ Array vs Linked Lists
  ▪ Create-Stack
  ▪ Push
  ▪ Pop
  ▪ Stack-Full
  ▪ Stack-Empty

◆ Implementation is easy, Very efficient : O(1)

◆ What about multiple stacks?
Applications : Stack

- Evaluation of Arithmetic Expressions
  - $3+2$, $3+5\times2$, $6/2-3+4\times2$, $$(2/(8\%4+(3\times5))\times(7-3))$$, . . .

- Parsing (Pattern Matching)
  - $a^n b^n$, $a^{2n} b^n$, palindromes, . . .

- Function Calls/Returns
  - Call function A, call B, Call C; How return?

- Maze Problem

- Depth First Search
Queue

◆ A linear List with front and rear.
◆ All insertions (enqueue) : rear,
  All deletions (dequeue) : front
◆ All items except front and rear are invisible.
◆ FIFO (First-In First-Out)
Implementing Queue

- Array vs Linked Lists
  - Create-Queue
  - Insert
  - Delete
  - Queue-Full
  - Queue-Empty
- For array, implementation is not so easy: $O(n)$
  - Use Circular Queue: $O(1)$
- What about multiple queues?
Applications : Queue

- Key board Data Buffers
- Job Processing (printer, CPU processor) : FCFS
- Breadth First Search
- Categorizing data into groups
- Waiting times of customers at call center
- Deciding # of cashiers at super market
- Traffic Analysis
Trees

Tree
- A non-linear list with nodes
- A special node: Root
- Parent: Child = 1:m relationship
- Leaf node: Node with no child
- Connected
- Acyclic Graph

Binary Tree
- Every node has at most 2 children. (0, 1, or 2)
- Order of children is important.
- Connected
- Acyclic Graph
What is height (h) of a binary tree
- \( n \) : #nodes of a binary tree;
- \( h \leq n \leq 2^h - 1 \)
- Thus, \( \log_2(n+1) \leq h \leq n \)
- \( n = 1,000? \ n = 1,000,000? \)

Question : What kind of trees do you prefer?
Implementing Binary Trees

- **Arrays**
  - 1-D array: A[ ]
  - Parent[i] = i/2, Lchild[i] = 2i, Rchild[i] = 2i+1

![Binary Tree Diagram]

- **Linked Lists**
  - Two links for each node
  - Lchild, RChild

- How many memories needed?

- How about trees? Array vs. LL?
Applications: Trees

- Hierarchical Information

- Tree Traversals: INORDER, PREORDER, POSTORDER

- Internal Searching: BST, AVL Tree, Red/Black Tree, 2-3 Tree, ..

- External Searching: B Tree, B+ Tree, ..

- Decision Trees: Classifications

- Min/Max Heaps
**Graphs**

**Graph**: $G = (V, E)$
- $V$: a (non-empty) set of vertices
- $E$: a set of edges $\subseteq (V \times V)$
- Undirected: $(u, v) = (v, u)$
- Directed: $(u, v) \neq (u, v)$
Implementing Graph

◆ **Adjacency Matrix** : $O(n^2)$
  - 2-D array : $A[n, n]$ ($n$ : #vertices)
  - $A(i, j) = 1$ if vertex $i$ and $j$ are adjacent
    - $= 0$ otherwise

(a) Adjacency matrix for non-directed graph

(b) Adjacency matrix for directed graph

◆ **Adjacency List** : $O(n + e)$
  - Each node consists vertex and link.
  - For each linked list $i$, it contains vertices adjacent from vertex $i$
Applications: Graphs

- Depth First Search, Breadth First Search
- Connectivity
- Minimal Spanning Trees
- Articulation Points
- Topological Sorting
- Activity On Vertex (AOV) Networks
- Activity On Edge (AOE) Networks
Exercise: Searching

- Given n numbers, find an arbitrary number X;
  Design an efficient data structure; Search, Insert, Delete;
  - Array: Unordered
  - Array: Ordered (Too much burden!!)
  - Binary Search Tree
  - AVL Tree
  - Quad, Octal, ... Tree
  - B-Tree (External Searching)
**Binary Search Tree (BST)**

1. **BST:**
   - Binary Tree
   - Every node’s key K is
     1. larger than all keys in its left subtree
     2. smaller than all keys in its right subtree.

2. Search/Insert/Delete: Time
   - Ex: Search(11), Search(18), Insert(18), ...

3. Height (h) of a BST:
   - Height(h) = number of edges on longest path from root to leaf.
Performance: BST

Average Case:
O(log₂n)

Worst Case:
O(n)
Improving Worst Case

- Basic Idea: **Balanced + Many Children**
  - Binary Search Tree
  - AVL Tree
  - 2-3-4 Tree
  - Quad, Octal Tree
  - B-Tree (External Searching)
Performance Comparison: Searching

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered Array</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Ordered Array</td>
<td>O(log₂n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>O(n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>O(log₂n)</td>
<td>O(log₂n)</td>
</tr>
<tr>
<td>2-3-4 Tree</td>
<td>O(log₂~₄n)</td>
<td>O(log₂~₄n)</td>
</tr>
<tr>
<td>B Tree (External)</td>
<td>O(log₁₃₃n)</td>
<td>O(log₁₃₃n)</td>
</tr>
</tbody>
</table>
Exercise: Searching Maximum Value

Given n numbers, find a maximum number X;
Application: Priority Queue

Design an efficient data structure; Search, Insert, Delete;

- Array: Unordered
- Array: Ordered
- Binary Search Tree
- Max Heap
**Max Heap**

**Max Heap:**

1. Complete binary tree
2. Value of each node is **no smaller** than its children’s values.

Note: **Root** of a max heap always has the **largest** value.
Performance: Max Heap

- Insert: $O(\log_2 n)$
- Delete: $O(\log_2 n)$

- Insert: 40, 45, ...
- Delete, Delete, ...

![Max Heap Diagram]

15
20
25
18
12
16
28
30
10
9
15

## Performance Comparison: Find Max

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unordered Linked list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Ordered Linked list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Max Heap</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
</tbody>
</table>
Constructing Algorithms

- Constructing Algorithm: Two Methods
  1. **Iteration**
     - while-loop, for-loop, repeat-until, . . .
     - Conventional Methods
  2. **Recursion**
     - Defined by calling itself.
     - Mostly based on divide and conquer
     - Simple, concise, high readability

- For every iterative algorithm, there exists an equivalently recursive algorithm; The reverse also is true.
Example: Factorial Number

**Iteration**
- Mathematical

\[
\text{Factorial} (n) = \begin{cases} 
1 & \text{if } n = 0 \\
(n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 & \text{if } n > 0 
\end{cases}
\]

**Recursion**

\[
\text{Factorial} (n) = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot \text{Factorial} (n-1) & \text{if } n > 0 
\end{cases}
\]

**Algorithmic**

```c
int factorial (int n) {
    i = 1; result = 1;
    while (i <= n) {
        result = result * i;
        i++;
    }
    return (result);
}
```

```c
int Factorial (int n) {
    if (n == 0) return(1);
    else return (n*Factorial(n-1));
}
```
Designing Recursion

- Rules for designing a recursion
  1. Base case
     - Trivial case
     - Usually, $n = 0$ or $n = 1$
     - For Termination
  2. General case (= Recursive step)
     - Break down the problem into sub-problems which are the same, but smaller size.
     - Usually, $n > 0$ or $n > 1$
  3. Combine base case and general case.
Binary Search

- Find an integer X among n ( > 1 ) integers; list[n]
  (All integers are stored by increasing order: Sorted)

- Construct Binary Search algorithms by recursion;
  (Use 3 variables : mid, left, right)

1. Base Case : Termination Condition
   (1) X is found : ?
   (2) X is not found : ?

2. General Case : Break a list into small size
   (1) X is in the first half (list[mid] > X) : ?
   (2) X is in the second half (list[mid] < X) : ?
Binary Search

**Bin-Search** (list[], X, left, right)

```c
int mid;
if (left <= right) {
    mid = (left + right)/2;
    if  X < list[mid], Bin-Search(list [], X, left, mid–1);
    else if  X == list[mid], return(mid);
    else, Bin-Search(list[], X, mid+1, right);
}
```

◆ What is time complexity? \( T(n) = T(n/2) + 1 \)
Binary Tree Traversal

- We want to visit every node in a binary tree.
  - **INORDER**: Left, Visit, Right (LVR)
  - **PREORDER**: Visit, Left, Right (VLR)
  - **POSTORDER**: Left, Right, Visit (LRV)

```plaintext
INORDER(p)
if (p != NULL)
    INORDER(p->Lchild)
    print(ptr->data)
    INORDER(p->Rchild)

POSTORDER(p)
if (p != NULL)
    PREORDER(p->Lchild)
    PREORDER(p->Rchild)
    print(ptr->data)
```
Computing $X^N$

```c
POWER (int X, int N)
    if (N == 0) return 1;
    else {
        factor = POWER(X, N/2)
        if  N%2 == 0  return  factor*factor
        else  factor*factor*X
    }
```

$X^N = (X^{N/2} \times X^{N/2})$ if $N$ : even

$X^N = (X^{N/2} \times X^{N/2}) \times X$ if $N$ : odd

- $N = 8; \ 2^8 = 2^4 \times 2^4, \ 2^4 = 2^2 \times 2^2, \ 2^2 = 2^1 \times 2^1$
- $N = 9; \ 2^9 = 2^4 \times 2^4 \times 2^1, \ 2^4 = 2^2 \times 2^2, \ 2^2 = 2^1 \times 2^1$
Towers of Hanoi

- **Base case**: \( n = 1 \)
  : Move 1 disk from source to dest

- **General case**: \( n > 1 \)
  (1) Move \((n - 1)\) disks from source to aux: (Use des as aux)

  (2) Move \((n - 1)\) disks from aux to des: (Use source as aux)

```c
void towers(int n, char source, char dest, char aux)
{
    if (n == 1) // base case
        printf("Move from %c to %c", source, dest);
    else { // general case
        towers(n - 1, source, aux, dest);
        printf("Move from %c to %c", source, dest);
        towers(n - 1, aux, dest, source);
    }
}
```

Recursion is Inefficient . . .

Which algorithm is more efficient?

<table>
<thead>
<tr>
<th>Iterative version</th>
<th>Recursive version</th>
</tr>
</thead>
<tbody>
<tr>
<td>int factorial (int n)</td>
<td></td>
</tr>
<tr>
<td>{ i = 1;</td>
<td></td>
</tr>
<tr>
<td>result = 1;</td>
<td></td>
</tr>
<tr>
<td>while (i &lt;= n)</td>
<td></td>
</tr>
<tr>
<td>{ result = result * i;</td>
<td></td>
</tr>
<tr>
<td>i++;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return (result);}</td>
<td>int Factorial (int n)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>if ( n == 0 ) return(1);</td>
<td></td>
</tr>
<tr>
<td>else return (n * Factorial (n - 1));</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Pros/Cons: Recursion

- **Pros/Cons**
  (+) Coding is simple, concise, clear.
  (+) Implementation is hidden;
  (+) High understandability, readability.
  (-) Space Overhead
  (-) Time Overhead

- **When do we need a recursion?**
  Do **not** use a recursion if the answer of the questions is ‘no’:

1. Is the algorithm naturally suited to recursion?
2. Is the recursive solution shorter and more understandable?
3. Does the recursive solution run within acceptable time and space?
Algorithm Design Techniques

- Brute Force
- Greedy method
- Divide and Conquer
- Dynamic Programming
- Backtracking
**Brute Force**

- A straightforward approach to; It tries to find all possible searching spaces.

- Easiest approach and useful for solving small size of a problem.

- Exhaustive search: May be exponential!

- Examples:
  - Computing $a^n$ (by multiplying $a \times a \times \ldots \times a$)
  - Selection Sort, Bubble Sort
  - Shortest Paths
  - Sequential search
Greedy Method

- At each solving step, choose the choice what it looks best; The choice must be locally optimal. Can’t see the global solution.

- Making the locally optimal choice at each stage with the hope of finding a global optimum. For example, road driving, card playing, . . .

- This method always does not give optimal solution, but it works for many problems in a reasonable time.

- Examples :
  - Minimal Spanning Tree
  - Shortest Paths
  - Fractional Knapsack
  - Huffman Coding
**Spanning Tree**

- **Spanning tree** $G'$ is a subgraph of a graph $G$ such that
  1. $V(G') = V(G) = n$ ($n : \# \text{ vertices}$)
  2. $G'$ is connected.
  3. $G'$ has $(n - 1)$ edges.
  4. If we add an edge into $G'$, then a cycle is generated.
  5. If we delete an edge from $G'$, then disconnected.

![Graph G and Some Spanning Trees $G'$ of G](image)
**Minimal Spanning Tree (MST)**

**Weighted Graph (G)**

![Graph](image)

**MST (G')**

![MST](image)

- **MST** is a spanning tree with minimum total weight.
- **Greedy Method** : (Kruskals’s algorithm : O(elog_2e))
  - (1) At each step, choose an edge with smallest weight.
  - (2) If the selected edge creates a cycle, then discard it.
  - (3) Repeat (1), (2); If sum of total edges are (n – 1), then done!
Divide and Conquer

- **Divide** a problem into many **smaller** sized sub-problems.

- Independently solve each sub-problem and then **combine** the sub-instance solutions to yield a solution for the original problem.

- The size of the problem is usually reduced by a factor (e.g., half the input size).

- **Examples**:
  - Binary Search
  - Quick Sort
  - Merge Sort
  - Strassen’s Matrix Multiplication
  - Computing $a^n$
Quick Sort (Top 10 algorithms in 20th Century)

Given a list of $n$ elements (e.g., integers):

- Pick one element to use as pivot.
- Partition elements into two sub-lists:
  - Left sub-lists $L$: Elements less than or equal to pivot
  - Right sub-lists $R$: Elements greater than pivot
- Recursively sort sub-list $L$ and $R$
- Combine the results
Quick Sort

Quicksort (list[], int left, right)
Partition; (list[], pivot)
Quicksort (list, left, j-1);
Quicksort (list, j+1, right);

Assume: Pivot is chosen as median of three.
Quick Sort: Time Complexity

◆ Worst Case
  - When the sub-lists are completely biased
  - Pivot is chosen as a smallest (largest) key for each split
  - $T(n) = T(n - 1) + c \cdot n$
  - $O(n^2)$
  - Rarely happens

◆ Average Case
  - When the sub-lists are likely balanced
  - Pivot is chosen as a random or median of three
  - $T(n) = 2 \cdot T(n/2) + c \cdot n$
  - $O(n \cdot \log_2 n)$
  - Fastest known sorting algorithm in practice
Dynamic Programming

- One drawback of "Divide and Conquer" is that the same computations repeatedly for identical sub-problems may arise.

- Dynamic Programming can avoid this drawback by defining the recurrence relation.

- Solve small sized sub-problems and store its result for later.

- The intermediate result can be reused for bigger problem.

- Examples:
  - Fibonacci Number
  - Warshall Algorithm
  - All Pairs Shortest Paths
  - 0/1 Knapsack
  - Matrix Chain Products
All pairs shortest paths

◆ Given a directed graph G with n vertices, find the shortest paths between every pairs of vertices

◆ Brute Force Approach:

◆ Dynamic Approach: Construct solution through series of matrices using increasing subsets of vertices allowed as intermediate.
All pairs shortest paths

- We define as $D^k[i,j]$ as: length of the shortest path from $i$ to $j$ without going through any vertex greater than $k$.
  - Without going through $k$: $D^{k-1}[i,j]$
  - Going through $k$: $D^{k-1}[i,k] + D^{k-1}[k,j]$

$$D^k[i,j] = \min \{D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j]\}$$

- Our goal: $k = n$; Compute $D^n[i,j]$ for every pair of vertices $i, j$ where $i, j, k$ in $[1, \ldots, n]$
All pairs shortest paths

- Compute $D^4[i, j]$ for every pair of vertices $i, j$;

\[ D^0 : \]
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & \infty & 3 & \infty \\
2 & 2 & 0 & \infty & \infty \\
3 & \infty & 7 & 0 & 1 \\
4 & 6 & \infty & \infty & 0 \\
\end{array}
\]

\[ D^1 : \]
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & \infty & 3 & \infty \\
2 & 2 & 0 & \text{\color{red}5} & \infty \\
3 & \infty & 7 & 0 & 1 \\
4 & 6 & \infty & \infty & 0 \\
\end{array}
\]

\[ D^2 : \]
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & \infty & 3 & \infty \\
2 & 2 & 0 & 5 & \infty \\
3 & \text{\color{red}9} & 7 & 0 & 1 \\
4 & 6 & \infty & 9 & 0 \\
\end{array}
\]

\[ D^3 : \]
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & \text{\color{red}10} & 3 & 4 \\
2 & 2 & 0 & 5 & 6 \\
3 & 9 & 7 & 0 & 1 \\
4 & 6 & \text{\color{red}16} & 9 & 0 \\
\end{array}
\]

\[ D^4 : \]
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 10 & 3 & 4 \\
2 & 2 & 0 & 5 & 6 \\
3 & 7 & 7 & 0 & 1 \\
4 & 6 & 16 & 9 & 0 \\
\end{array}
\]

- For example, $D^1[2, 3] = \min \{D^0[2, 3], D^0[2, 1]+D^0[1, 3]\}$
  $= \min \{\infty, 2+3\} = 5$
All pairs shortest paths

- **Floyd Algorithm**

```plaintext
for (k=1; k<=n; i++)
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      D^k[i, j] = min {D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]}
```

- **Time Complexity**: $O(n^3)$
- **Space Complexity**: $O(n^2)$
- **Note**: Works on graphs with negative edges but without negative cycles.
Backtracking

- A sort of brute force approach, but additional condition that only the possible candidate solutions are considered.

- A systematic searching method by pruning searching spaces; This is to avoid unnecessary efforts as early as possible.

- Upon failure, we can go back to the previous choice simply by returning a failure node.

- Backtracking vs. DFS

- Examples:
  - Maze Problem
  - N-Queens Problem
  - Graph Coloring
  - Hamiltonian Cycle
  - Data Mining: Apriori Algorithm
Backtracking

FIGURE 3-17 Backtracking Example

<table>
<thead>
<tr>
<th>At 4</th>
<th>At 6</th>
<th>At 1st end</th>
<th>At 2nd end</th>
<th>At 3rd end</th>
<th>At goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>B9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B12</td>
<td>3</td>
<td>B12</td>
<td>B12</td>
<td>B12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>goal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
In backtracking, we explore each node, as follows:

To explore node N:
1. If N is a goal node, return “success”
2. If N is a leaf node, return “failure”
3. For each child C of N,
   3.1. Explore C
      3.1.1. If C was successful, return “success”
4. Return “failure”
Hard Problems

◆ So far, many problems can be solved by efficient algorithms.

◆ In other respect, for many problems, any efficient algorithms have not been found; What’s worse, for such problems, we can’t even tell whether or not an efficient solution might exist.

◆ Programmers : Why can not find such efficient algorithms?
Theoreticians : Why can not find any reason why these problems should be difficult?

◆ Consider the following problems;
  ▪ Easy : Is there a path from x to y with weight $\leq M$
    - Shortest Path : $O(n)$
  ▪ Hard(?) : Is there a path from x to y with weight $\geq M$
    - Longest Path : $O(2^n)$
Hard Problems

- **$P$** Problems
  - Can be solved by deterministic algorithms in polynomial time.
  - Can be solved with efficient amount of time.
  - Searching, Soring, . . .

- **$NP$** Problems
  - Can be solved by non-deterministic algorithms in polynomial time.
  - For many problems, only exponential time algorithms are known.
    (Deterministic polynomial time algorithms are not known (so far).)
  - Can not be solved with efficient amount of time.
  - Satisfiability, Graph Coloring, . . .

- Relationship between $P$ and $NP$
  - Clearly, $P \subseteq NP$ (Any problem in $P$ is in $NP$)
  - The biggest open problem in Computer Science;
    - Is $P \subset NP$ or $P = NP$?
Unsolvable (Undecidable) Problems

Is every problem is solvable?
- The number algorithms is countably infinite.
- The number of problems is un-countably infinite.
- There exist some problems not solvable by any algorithms.
- There exist infinite number of problems not solvable by computers.
- Turing-Undecidable

Examples
- Post Correspondence Problem (PCP)
- Halting Problem
- Ambiguity Problem
- . . . . . .
Conclusions To Remember

◆ Lesson 1 :
  Good algorithms are better than super computers.

◆ Lesson 2 :
  Good algorithms are better than good algorithms.

◆ Lesson 3 :
  Good data structures are essential for good algorithms.

◆ Lesson 4 :
  Try to remember a few well known algorithms.

◆ Lesson 5 :
  Try to learn programming languages and exercise coding.