Introduction

• What is Algorithm?
• Modern Theory of Algorithms
  - Algorithm manner from Ancient to Modern times
• Serial and Parallel Computing
• Mathematics in analysis
• Recursion

• Find the square root of 7
• How far are you away from the exact value?
Ancient Algorithms

- Babylonians knew how to approximate square roots (500 B.C.)
- Euclid`s algorithm (300 B.C.) still used today to find GCD.
- Newton’s method (1600’s) generalizes this to find zeroes of polynomial
- “Numerical” algorithms
Modern Theory of Algorithm

- Difference Engine $\Rightarrow$ Analytic engine
- Turing Machine
- Instruction usage
- Data Structures
- Now

Serial, Parallel processors, Computing in Space travel, robotics, graphics, simulations of complex systems and medical analysis.
Mathematical Review *

- **Properties of logarithms:**
  \[
  \begin{align*}
  \log_b(xy) &= \log_b x + \log_b y \\
  \log_b (x/y) &= \log_b x - \log_b y \\
  \log_b x^a &= a \log_b x \\
  \log_b a &= \log_x a / \log_x b
  \end{align*}
  \]

- **Properties of exponentials:**
  \[
  \begin{align*}
  a^{(b+c)} &= a^b a^c \\
  b^c &= a^{\log_a b} \\
  b^c &= a^{c \log_a b}
  \end{align*}
  \]

- **Properties of summation:**
  \[
  \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}
  \]
Recursion

- **Recursion** is a programming technique in which a method calls itself.
- Each invocation of the method has its own set of local variables.
- Recursion can often simplify a problem.
- It is not always the most efficient solution.
- Any recursive problem can be solved without recursion.
Factorial Example

- \( n! = n \times (n-1) \times \ldots \times 2 \times 1 \)
- A non-recursive solution:

```java
public long factorial(int n) {
    long result = 1;
    for (int i=n; i>1; i--)
        result *= i;
    return result;
}
```

- A recursive solution

```java
public long factorialR (int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorialR (n - 1);
}
```

- Each recursive call must make the problem “smaller”
- There must be special, non-recursive handling of the simplest case(s)
- Note that \( n! = n \times (n-1)! \)
Towers of Hanoi

• The general case:
  – The puzzle consists of N disks and three poles: A (the source), B (the destination), and C (the spare)
• Consider, for a second the simpler case:
  – The puzzle consists of 3 disks and three poles: A (the source), B (the destination), and C (the spare)
Guidelines For Software Design

• Formulate the problem
• Choose data types for modeling
• Use already solved algorithms, find algorithms for rest
• Verify its correctness
• Analyze its efficiency
• Decide whether to improve
Chapter 2 *
Elementary Data Structures

- Stacks, Queues, Lists, Trees, Priority Queues
Abstract Data Types (ADTs) *

An abstract data type (ADT) is an abstraction of a data structure.

An ADT specifies:
- Data stored
- Operations on the data
- Error conditions associated with operations

Play very important role in algorithm design.
List ADT *

• Linked list
  – linear collection of self-referential class objects, called *nodes*, connected by pointer *links*
  – accessed via a pointer to the first node of the list
  – subsequent nodes are accessed via the link-pointer member
  – the link pointer in the last node is set to null to mark the list’s end

• Use a linked list instead of an array when
  – the number of data elements is unpredictable
  – the list needs to be sorted
The Stack ADT *

- stack – new nodes can be added and removed only at the top
  - similar to a pile of dishes
  - last-in, first-out (LIFO)
  - constrained version of a linked list
- push
  - adds a new node to the top of the stack
- pop
  - removes a node from the top

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment
The Queue ADT *

- queue – similar to a supermarket checkout line
  - *first-in, first-out (FIFO)*
  - nodes are removed only from the *head*
  - nodes are inserted only at the *tail*

- The insert and remove operations are known as enqueue and dequeue
- Useful in computing
  - Print spooling, packets in networks, file server requests

Applications of Queues

- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)
  - Multiprogramming
Trees *

- Tree nodes contain two or more links
- Binary trees
  - all nodes contain two links
  - The *root node* is the first node in a tree.
  - Each link in the root node refers to a *child*
  - A node with no children is called a *leaf node*
Binary Search Tree *

- values in left subtree less than parent
- values in right subtree greater than parent
- facilitates duplicate elimination
- fast searches - for a balanced tree, maximum of $\log n$ comparisons
**Tree Traversals**

– **inorder traversal** of a binary search tree prints the node values in ascending order
  1. Traverse the left subtree with an inorder traversal.
  2. Process the value in the node (i.e. print the node value).
  3. Traverse the right subtree with an inorder traversal.

– **preorder traversal:**
  1. Process the value in the node.
  2. Traverse the left subtree with a preorder traversal.
  3. Traverse the right subtree with a preorder traversal.

– **postorder traversal:**
  1. Traverse the left subtree with a postorder traversal.
  2. Traverse the right subtree with a postorder traversal.
  3. Process the value in the node.
Complete & Perfect Binary Tree

Full (Perfect) Binary Trees
• A binary tree where each node is either a leaf or is an internal node with exactly two non-empty children.
• That means, a node is allowed to have either none or two children (but not one!)

Complete Binary Trees
• A binary tree whereby if the height is \(d\), and all levels, except possibly level \(d\), are completely full. If the bottom level is incomplete, then it has all nodes to the left side.
  - That is the tree has been filled in the level order from left to right.

![Perfect (Full) Binary Tree](image1)
![Not Full Binary Tree](image2)
Multi-Way Search Tree *

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores \( d - 1 \) key-element items \((k_i, o_i)\), where \( d \) is the number of children
  - For a node with children \( v_1, v_2, \ldots, v_d \) storing keys \( k_1, k_2, \ldots, k_{d-1} \)
    - keys in the subtree of \( v_1 \) are less than \( k_1 \)
    - keys in the subtree of \( v_i \) are between \( k_{i-1} \) and \( k_i \) \((i = 2, \ldots, d - 1)\)
    - keys in the subtree of \( v_d \) are greater than \( k_{d-1} \)
  - The leaves store no items and serve as placeholders
Chapter 3
Design & Analysis of Algorithms

Input → Algorithm → Output
THE COMPLEXITY OF ALGORITHMS

To analyze the complexity of an algorithm:
- identify a basic operation
- count how many times the algorithm performs this operation

Define a notion of size for an input instance:
- let $S_n$ be the set of all inputs of size $n$ to an algorithm
- $\tau(I)$ be the number of basic operations performed when the algorithm is executed with input $I$. 
Average case analysis *

- **Drawbacks**
  - Based on a probability distribution of input instances
  - How do we know if distribution is correct or not?

- Usually more complicated to compute than worst case running time
  - Often worst case running time is comparable to average case running time (see next graph)
  - Counterexample to above: Quicksort
Algorithm Complexity as a function of size n

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input of size n</th>
<th>Basic operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching a list</td>
<td>lists with n elements</td>
<td>comparison</td>
</tr>
<tr>
<td>Sorting a list</td>
<td>lists with n elements</td>
<td>comparison</td>
</tr>
<tr>
<td>Multiplying two matrices</td>
<td>two n-by-n matrices</td>
<td>multiplication</td>
</tr>
<tr>
<td>Prime factorization</td>
<td>n-digit number</td>
<td>division</td>
</tr>
<tr>
<td>Evaluating a polynomial</td>
<td>polynomial of degrees n</td>
<td>multiplication</td>
</tr>
<tr>
<td>Traversing a tree</td>
<td>tree with n nodes</td>
<td>accessing a node</td>
</tr>
<tr>
<td>Towers of Hanoi</td>
<td>n disks</td>
<td>moving a disk</td>
</tr>
</tbody>
</table>

- A comparison-based algorithm for searching or sorting a list is based on
  - making comparisons involving list elements
  - then making decisions based on these comparisons.
LINEAR SEARCH

Pseudocode

function LinearSearch ( L[1:n], X )
    for i := 1 to n do
        if X := L[i] then
            return (i)
        end if
    end for
    return (0)
end LinearSearch

Complexity

B (n) = 1 ........... when X = L[1]
W (n) = n ........... when X = L[n]
A (n) = ( n+1 ) / 2
**Binary Search**

**Pseudocode**

```plaintext
function BinarySearch ( L[1:n], X )
    Found := .false.
    low := 1
    high := n
    while .not. Found .and. low ≤ high do
        Mid := floor((low+high)/2)
        case
            :X < L[mid] : high := mid - 1
            :otherwise: low := mid + 1
        endcase
    endwhile
    if Found then
        return ( mid )
    else
        return ( 0 )
    endif
end BinarySearch
```

**Complexity**

- \( B(n) = 1 \) ............ when \( X = L[mid] \)
- \( W(n) = \log_2 (n+1) \) which is equal to the longest string of midpoints ever generated
Finding the Max and Min Elements in a list

**TRADITIONAL WAY**

**Pseudocode**

```plaintext
function Max ( L[1:n] )

    MaxValue := L[1]
    for i:= 2 to n do
        if L[i] > MaxValue then
            MaxValue := L[i]
        endif
    endfor

    return (MaxValue)

d function Max
```

**Complexity**

\[ B(n) = W(n) = A(n) = n - 1 \]
SMARTER WAY

PSEUDO CODE

procedure MaxMin ( L[1:n], MaxValue, MinValue)
  if even(n) then
    call M&M (L[1], L[2], MaxValue, MinValue)
    for i:= 3 to n-1 by 2 do
      call M&M ( L[i], L[i+1], b, a )
      if a < MinValue then MinValue := a endif
      if b < MaxValue then MaxValue := b endif
    endfor
  else
    MaxValue := L[1]; MinValue:= L[1];
    for i:= 2 to n-1 by 2 do
      call M&M ( L[i], L[i+1], b, a )
      if a < MinValue then MinValue := a endif
      if b < MaxValue then MaxValue := b endif
    endfor
  endif
end MaxMin

procedure M&M
  if A ≥ B then
    MaxValue := A
    MinValue := B
  else
    MaxValue := B
    MinValue := A
  endif
end M&M

COMPLEXITY

B(n) = W(n) = A(n) = \left\lceil \frac{3n}{2} \right\rceil - 2
**Pseudocode**

```plaintext
procedure InsertionSort ( L[1:n] )
    for i := 2 to n do
        Current := L[i]
        position := i-1
        while position ≥ 1 .and. Current < L[position] do
            L[position+1] := L[position]
            position := position - 1
        endwhile
        L[position+1] := Current
    end for
end InsertionSort
```

**Complexity**

- \( B(n) = n-1 \) if the list is already sorted in nondecreasing order
- \( W(n) = \frac{n(n-1)}{2} \) if the list is in strictly decreasing order
**Insertion Sort** is
- easy to program
- not efficient for large $n$
- very efficient on nearly sorted large lists
- is an on-line sorting algorithm, the entire list is not input to the algorithm in advance; elements are added over time
- is a stable sorting algorithm, it maintains the relative order of repeated elements
A Glimpse of Lower Bound Theory

- **Lower bound** for a problem: Minimum complexity that can be achieved by any algorithm for solving that problem.

- **Optimal Algorithm** for a problem: An algorithm whose complexity equals to the lower bound for that problem.

- **An adjacent-key comparison-based sorting algorithm** is one in which comparisons b/w list elements are made only b/w elements that occupy adjacent positions.

  - A lower bound for the $W(n)$ of any adjacent-key comparison-based sorting algorithm is $n(n-1)/2$.

  - Lower bound for finding the maximum element in a list of size $n$ has $W(n)$, $B(n)$, $A(n)$ all equal to $n-1$. 
Asymptotic Behaviour of Functions

- An exact computation of worst-case running time can be difficult
  - Function may have many terms:
    \[ 4n^2 - 3n \log n + 17.5 n - 43 n^{2/3} + 75 \]

Simplifications

- Ignore constants
  - \[ 4n^2 - 3n \log n + 17.5 n - 43 n^{2/3} + 75 \] becomes \[ n^2 - n \log n + n - n^{2/3} + 1 \]

Asymptotic Efficiency

- \[ n^2 - n \log n + n - n^{2/3} + 1 \] becomes \( n^2 \)

End Result: \( \Theta(n^2) \)
Asymptotic Analysis

We focus on the infinite set of large $n$ ignoring small values of $n$.

Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
O \( (g (n)) \) is a set of functions.

However, we will use one-way equalities like

\[ n = O(n^2) \]

This really means that function \( n \) belongs to the set of functions \( O(n^2) \)

Incorrect notation: \( O(n^2) = n \)
Big Oh” Notation

\( O(g(n)) = \{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } \ n \geq n_0, \ 0 \leq f(n) \leq c \ g(n) \} \)

“Big Omega” Notation

\( \Omega(g(n)) = \{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } \ n \geq n_0, \ 0 \leq f(n) \geq c \ g(n) \} \)

“Big Theta” Notation

\( \Theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } \ n \geq n_0, \ c_1 \ g(n) \leq f(n) \leq c_2 \ g(n) \} \)
$O(g(n))$

Upper Bound on $f(n)$

$\Omega(g(n))$

Lower Bound on $f(n)$
\( \Theta(g(n)) \)

\( c_1 \times g(n) \) is an *Upper Bound* on \( f(n) \)

\( c_2 \times g(n) \) is a *Lower Bound* on \( f(n) \)
$O(f(n))$ and $\Omega(g(n))$
Common time complexities

- **O(1)**: constant time
- **O(log n)**: log time
- **O(n^{1/2})**: log time
- **O(n)**: linear time
- **O(n log n)**: log linear time
- **O(n^2)**: quadratic time
- **O(n^3)**: cubic time
- **O(2^n)**: exponential time
- **O(n!)**: factorial time

Better: UP
Worse: DOWN
## Common Complexity Functions

<table>
<thead>
<tr>
<th>Complexity</th>
<th>(10)</th>
<th>(20)</th>
<th>(30)</th>
<th>(40)</th>
<th>(50)</th>
<th>(60)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>(1 \times 10^{-5}) sec</td>
<td>(2 \times 10^{-5}) sec</td>
<td>(3 \times 10^{-5}) sec</td>
<td>(4 \times 10^{-5}) sec</td>
<td>(5 \times 10^{-5}) sec</td>
<td>(6 \times 10^{-5}) sec</td>
</tr>
<tr>
<td>(n^2)</td>
<td>(0.0001) sec</td>
<td>(0.0004) sec</td>
<td>(0.0009) sec</td>
<td>(0.016) sec</td>
<td>(0.025) sec</td>
<td>(0.036) sec</td>
</tr>
<tr>
<td>(n^3)</td>
<td>(0.001) sec</td>
<td>(0.008) sec</td>
<td>(0.027) sec</td>
<td>(0.064) sec</td>
<td>(0.125) sec</td>
<td>(0.216) sec</td>
</tr>
<tr>
<td>(n^5)</td>
<td>(0.1) sec</td>
<td>(3.2) sec</td>
<td>(24.3) sec</td>
<td>(1.7) min</td>
<td>(5.2) min</td>
<td>(13.0) min</td>
</tr>
<tr>
<td>(2^n)</td>
<td>(0.001) sec</td>
<td>(1.0) sec</td>
<td>(17.9) min</td>
<td>(12.7) days</td>
<td>(35.7) years</td>
<td>(366) cent</td>
</tr>
<tr>
<td>(3^n)</td>
<td>(0.59) sec</td>
<td>(58) min</td>
<td>(6.5) years</td>
<td>(3855) cent</td>
<td>(2 \times 10^8) cent</td>
<td>(1.3 \times 10^{13}) cent</td>
</tr>
<tr>
<td>(\log_2 n)</td>
<td>(3 \times 10^{-6}) sec</td>
<td>(4 \times 10^{-6}) sec</td>
<td>(5 \times 10^{-6}) sec</td>
<td>(5 \times 10^{-6}) sec</td>
<td>(6 \times 10^{-6}) sec</td>
<td>(6 \times 10^{-6}) sec</td>
</tr>
<tr>
<td>(n \log_2 n)</td>
<td>(3 \times 10^{-5}) sec</td>
<td>(9 \times 10^{-5}) sec</td>
<td>(0.0001) sec</td>
<td>(0.0002) sec</td>
<td>(0.0003) sec</td>
<td>(0.0004) sec</td>
</tr>
</tbody>
</table>